

Memoria Proyecto de Sistemas Informáticos:

Huellas en la Tierra: Web social usando Google Maps

Autores:

García García, Daniel
Ramos Romero, José Luis

Profesor Director:

Alberto Verdejo

CURSO 2010/2011

Facultad de Informática

Universidad Complutense de Madrid

Autorización de difusión de contenidos

Autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, la presente memoria, así como el código y el resto del material que componen este proyecto.

García García, Daniel
Fdo.

Ramos Romero, José Luis
Fdo.

Resumen

Este proyecto surge de una idea: darle una función social a las herramientas que ofrece Google Maps.

Todos necesitamos en algún momento compartir una situación en el mapa con nuestros amigos o conocidos. Esta situación algunas veces es algo impersonal y no nos importa que sea de dominio público (ej.: la dirección de un restaurante), pero otras puede ser una información de carácter más privado (como la dirección de nuestro domicilio personal) y que es importante que sólo pueda ser vista por un grupo de personas que nosotros mismos elijamos. Aquí es cuando entra la idea de las "redes sociales", tan populares hoy en día, en las que hay permisos especiales dependiendo de la relación entre dos usuarios.

Llegados a este punto decidimos crear *huellasenlatierra.com*, una página web que reunirá las necesidades comentadas. Nuestra aplicación contará con un registro de usuarios para identificarse y poder crear las marcas o "huellas" de nuestros puntos de interés en el mapa, y dotarlas de privilegios para que sean visibles sólo por el propio usuario, por este y sus amigos o por todo el mundo.

Además se han añadido funcionalidades extra al mapa, como poder ver el clima en tiempo real, o las fronteras de las provincias, coloreadas en función del tiempo que haga en cada una.

This project arises from an idea: to give a social function to the tools that Google Maps offers.

All we at some time needed to share a situation in the map with our friends or acquaintances. This situation some times is something impersonal and it does not matter to us that it is of public dominion (ex.: the direction of a restaurant) but others, can more be a information of private character (like the direction of our personal address) and that is important that it only can be seen by a group of people who we ourself we choose. Here it is when the idea of the "social networks" enters, so popular nowadays, in which there are special permissions following the relation between two users. Arrived at this point, we decided to create "huellasenlatierra.com", a web page that will meet the commented needs.

Our application will count on a registry of users to identify itself and to be able to create the marks or "tracks" of our points of interest in the map, and to equip them with privileges so that they are visible only by the own user, by this and his friendly or by everybody.

In addition it is added functionalities extra to the map, like being able to see the climate in real time, or the borders of the provinces colored based on the time that does in each.

Palabras clave

- Servidor
- Página web
- Mapa interactivo
- *Google Maps*
- Punto de interés
- Privacidad
- Red Social
- PHP
- MySQL
- Ajax

Índice

Lista de abreviaturas y términos

- **Huella:** Es un marcador que indica una posición en el mapa. Este marcador tiene un nombre y una descripción que normalmente especifica qué podemos encontrar en esa ubicación. También denominamos muchas veces esta palabra como **punto de interés**.

- **Coordenada:** Es el par de números reales formado por la longitud y la latitud que indican la ubicación de un punto en el mapa.

- **Fuga de memoria:** Es un error de software que ocurre cuando un bloque de memoria reservada no es liberada. Comúnmente ocurre porque se pierden todas las referencias a esa área de memoria antes de haberse liberado.

- **Cron Job:** Es un proceso automatizado que se ejecuta en el servidor en intervalos de tiempo definidos por el usuario.

- **Transacción:** En un Sistema de Gestión de Bases de Datos (SGBD), una transacción es un conjunto de órdenes que se ejecutan formando una unidad de trabajo, es decir, en forma indivisible. Un *SGBD* transaccional es capaz de mantener la integridad de los datos, haciendo que estas transacciones no puedan finalizar en un estado intermedio.

- **Método *GET*:** Es la forma de enviar información (como por ejemplo, datos recogidos de un formulario) a una página web a través de su URL. Para ello, se añade el símbolo *?* al final de la misma y se insertan los parámetros, uno a uno, separados por el carácter *&*. Como muestra, la URL *http://www.ejemplo.com/ej.html?campo1=0?campo2=abc* contaría con dos parámetros *campo1* (con valor *"0"*) y *campo2* (igual a "*abc"*) y el código HTML del archivo *ej.html* recogería dichos parámetros mediante el método GET.

- **Método *POST*:** Este método se diferencia del anterior en que los datos son enviados por la entrada estándar STDIO sin necesidad de añadirlos a la URL de la página.

- ***Meteored*:** Es la página web a la que le solicitamos los datos meteorológicos actuales mediante las funciones que ofrece su API de acceso al tiempo.

Tecnologías empleadas

1. HTML y CSS

HTML (en inglés, *HyperText Markup Language*) es el lenguaje de marcado predominante para la elaboración de páginas web. Es usado para describir la estructura y el contenido en forma de texto, así como para complementar el texto con diferentes objetos como imágenes o animaciones. HTML se escribe en forma de "etiquetas", rodeadas por corchetes angulares (<,>) y puede describir, hasta un cierto punto, la apariencia de un documento. Además, puede incluir un script (por ejemplo Javascript), el cual puede afectar el comportamiento de navegadores web y otros procesadores de HTML.

Las hojas de estilo en cascada CSS (en inglés *Cascading Style Sheets*) es un lenguaje usado para definir la presentación de un documento estructurado escrito en HTML o XML (y por extensión en XHTML). El W3C (*World Wide Web Consortium*) es el encargado de formular la especificación de las hojas de estilo que servirán de estándar para los agentes de usuario o navegadores.

La idea que se encuentra detrás del desarrollo de CSS es separar la estructura de un documento de su presentación. Así pues, el uso de HTML+CSS era necesario a la hora de diseñar nuestra página web.

En cuanto a la versión de HTML utilizada, empleamos HTML4 para el desarrollo de la web. En la actualidad existe ya HTML5, que incluye algunas mejoras en relación con su antecesor. Una de ellas es la comodidad que aporta a la hora de usar algunas de las nuevas etiquetas, ahora hay etiquetas definidas para secciones típicas como el *header* (cabecera de la web) o el *footer* (el pie de página de la web). De esta forma la sintaxis es más sencilla:

Antes: `<div id="header">`
Ahora: `<header>`

Otra nueva ventaja es la forma de insertar videos, labor que ahora es increíblemente sencillo con HTML5, basta con usar la etiqueta `<video> … </video>`. Entre la etiqueta de apertura y la de cierre sólo habría que introducir la dirección del enlace del video.

Hasta este punto parece que esta nueva versión de HTML5 sólo aportaría mejoras a nuestro proyecto, pero también implicaría complicaciones. En concreto, la más significativa sería la compatibilidad. El HTML5 aún no es un estándar y, por tanto, no todos los navegadores lo tratan tan correctamente como deberían. Mientras la mayor parte de ellos, como *Firefox*, *Chrome* o *Safari*, lo procesan de

manera más o menos eficiente, el problema viene, como suele ser usual, con *Internet Explorer(por otra parte el más extendido)*.

La versión de *Internet Explorer* cuando comenzamos con el proyecto era la 8, la cual está probado que falla bastante con HTML5. La nueva versión (*Internet Explorer 9*) no era una realidad en aquel entonces ya que hasta hace unos días se encontraba en su versión beta. Según Microsoft, esta versión actualizada soluciona los problemas y acepta el nuevo HTML sin problemas.

En definitiva, como la versión 8 de *Internet Explorer* falla con HTML5 y, por desgracia, aún tiene una gran cuota de mercado que no podemos ignorar, hemos decidido usar HTML4, versión que aportará mayor estabilidad al proyecto, aunque se pierda comodidad de programación.

2. JavaScript y AJAX

Javascript es un lenguaje de programación interpretado, dialecto del estándar *ECMAScript.* Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.

Se utiliza principalmente en su forma del lado del cliente, implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas, aunque existe una forma de Javascript del lado del servidor (*Server-side Javascript* o SSJS).

AJAX, acrónimo de *Asynchronous JavaScript And XML* (en español, JavaScript asíncrono y XML), es una técnica de desarrollo web para crear aplicaciones web interactivas o RIA (*Rich Internet Applications*). Estas aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene la comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, lo que significa aumentar la interactividad, velocidad y usabilidad en las aplicaciones.

El uso de AJAX nos ha permitido poder interaccionar con la base de datos una vez cargada la web, de manera asíncrona. Además, la gran potencia que tiene nos ha permitido hacer un portal que se maneja, en gran parte, sin necesidad de recargar la página, lo que es una gran mejora para el usuario, tanto en comodidad como en rapidez.

Comentaremos detalles sobre su uso al hablar de la implementación.

3. PHP

PHP es un lenguaje de programación interpretado, diseñado originalmente para la creación de páginas web dinámicas. Es usado principalmente en interpretación del lado del servidor.

El uso de PHP nos ha permitido gestionar las sesiones de usuario e interactuar, a través del lenguaje SQL, con la base de datos para generar los documentos XML solicitados por las llamadas de AJAX.

4. JSON, XML y KML

XML (en inglés, *eXtensible Markup Language*) es un metalenguaje de etiquetas extensible desarrollado por el *World Wide Web Consortium* (W3C) que permite definir la gramática de lenguajes específicos. Por lo tanto, XML no es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades. Se puede usar en bases de datos, editores de texto, hojas de cálculo y casi cualquier cosa imaginable. Es una tecnología sencilla que tiene a su alrededor otras que la complementan y la hacen mucho más grande y con unas posibilidades mucho mayores. Tiene un papel muy importante en la actualidad ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil.

KML (del acrónimo en inglés *Keyhole Markup Language*) es un lenguaje de marcado basado en XML para representar datos geográficos en tres dimensiones. Fue desarrollado para ser manejado con *Keyhole LT*, precursor de *Google Earth*. Su gramática es similar a la de XML y es interpretado por *Google Maps* para superponer en los mapas diferentes objetos como marcas, líneas, polígonos e imágenes.

JSON, acrónimo de *JavaScript Object Notation*, es un formato ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos de JavaScript que no requiere el uso de XML.

La simplicidad de JSON ha dado lugar a la generalización de su uso, especialmente como alternativa a XML en AJAX. Una de las supuestas ventajas de JSON sobre XML como formato de intercambio de datos en este contexto es que es mucho más sencillo escribir un analizador semántico de JSON. En JavaScript, un texto JSON se puede analizar fácilmente usando el procedimiento `eval()`, lo cual ha sido fundamental para que JSON haya sido aceptado por parte de la comunidad de desarrolladores AJAX, debido a la ubicuidad de JavaScript en casi cualquier navegador web.

Si bien es frecuente ver JSON posicionado contra XML, también es frecuente el uso de JSON y XML en la misma aplicación. De hecho, en nuestro proyecto utilizamos JSON para interpretar documentos XML y obtener, en Javascript, la información contenida en ellos.

Además, para señalar las fronteras en el mapa de nuestra página web lo que hacemos es leer un documento KML donde se concentra toda la información que conforman los polígonos de las regiones, comunidades y países.

5. DOM

El *Document Object Model* o DOM es esencialmente una interfaz de programación de aplicaciones (API) que proporciona un conjunto estándar de objetos para representar documentos HTML y XML, un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos. A través del DOM, los programas pueden acceder y modificar el contenido, estructura y estilo de los documentos HTML y XML, que es para lo que se diseñó principalmente. El responsable del DOM, al igual que con XML, es el *World Wide Web Consortium* (W3C).

El DOM nos ha sido de gran utilidad en nuestro proyecto ya que nos ha permitido añadir y cambiar dinámicamente el contenido estructurado de Javascript.

6. SQL

El lenguaje de consulta estructurado o SQL (en inglés, *structured query language*) es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones en estas. Una de sus características es el manejo del álgebra y el cálculo relacional permitiendo efectuar consultas con el fin de recuperar, de una forma sencilla, información de interés de una base de datos, así como también hacer cambios sobre ella.

El uso de este lenguaje es básico en nuestra aplicación ya que nos permite acceder a nuestra base de datos y poder controlar de esta forma toda la información sobre los puntos de interés personales y públicos, el registro, *login* y *logout* de los usuarios, el manejo de las solicitudes de amistad y de los amigos existentes, la información metereológica de los diferentes lugares, etc.

7. API Google Maps

Una interfaz de programación de aplicaciones o API (del inglés *Application Programming Interface*) es el conjunto de funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción. Son usados generalmente en las bibliotecas.

La API de Google Maps es un servicio gratuito, disponible para cualquier sitio web que sea gratuito para el consumidor, que permitirá integrar un mapa con ciertas funcionalidades que Google nos ofrece y que se pueden combinar con las que nosotros queramos crear. Nos parece una parte muy interesante, por lo que, a continuación, hemos dedicado una sección entera para aprender su manejo y las herramientas que proporciona.

API Google Maps

El API de Google Maps fue lanzado en junio de 2005, haciendo oficialmente modificable casi cualquier aspecto de la interfaz original de Google Maps. Este API permite al usuario insertar las funciones más completas y la utilidad diaria de Google Maps en su propio sitio web y en sus propias aplicaciones, así como superponer sus datos sobre ellas.

En nuestro proyecto hemos utilizado la versión 2 de **Google Maps Javascript API**. De esta forma, la agilidad en el desarrollo de nuestro sitio web se ha visto favorecida ya que este API utiliza Javascript, un lenguaje de programación mundialmente extendido y con el que el programador se familiariza rápidamente. Cuando se comenzó con este proyecto, la versión 3 no estaba completamente desarrollada, por lo que se decidió utilizar la versión previa, que en aquel entonces estaba más extendida y se presentaba al desarrollador como la versión más estable. En la actualidad, la versión 3 está especialmente diseñada para proporcionar una mayor velocidad y se puede aplicar más fácilmente tanto a móviles como a las aplicaciones de navegador de escritorio tradicionales.

1. Carga del API y del mapa

El primer paso para poder utilizar el API de Google Maps es cargar dicho API. Para ello es necesario registrarse y obtener una **clave** que será válida para un **dominio único**, en nuestro caso, *huellasenlatierra.com*. De esta forma, Google Maps API nos permitirá insertar Google Maps en nuestras páginas web. La clave de API está conectada con una cuenta de Google, por lo que es necesario contar con una antes del proceso de registro. Una vez completado, Google nos proporcionará la clave única y ya estaremos en disposición de cargar el API en nuestra web. Esto lo haremos mediante el siguiente script:

```
<script src=http://maps.google.com/maps?file=api&v=2&key=1234&sensor=true
type="text/javascript"> </script>
```

La URL `http://maps.google.com/maps?file=api&v=2&key=1234` permite acceder a la ubicación del archivo JavaScript que incluye todos los símbolos y definiciones necesarios. Para acceder al API, la página debe contener una etiqueta `script` con la clave recibida anteriormente. En este ejemplo, la clave se muestra como `1234`. Vemos también que existe un parámetro `sensor` para indicar si la aplicación utilizará un sensor para determinar la ubicación del usuario. Este valor se define como `true` o `false`. En este ejemplo, hemos dejado el parámetro como `true`.

Para mostrar el mapa en nuestra página web debemos **reservar** un lugar para él. Esto se logra, generalmente, creando un elemento `div` con el identificador

que deseemos y obteniendo una referencia a este elemento en el modelo de objetos de documento (DOM) del navegador.

```
<div id="mapa" style="width: 500px; height: 300px"></div>
```

En este ejemplo, definimos un objeto `div` denominado `mapa` y definimos su tamaño mediante atributos de estilo. Si no especificamos de forma explícita un tamaño para el mapa (veremos más adelante cómo se hace), el mapa usará el tamaño del contenedor para definir su propio tamaño.

El **elemento fundamental** en cualquier aplicación del API de Google Maps es el propio mapa, representado por la clase de Javascript denominada `GMap2`. Cada objeto de esta clase define un único mapa en una página. Para crear una nueva instancia de esta clase usamos el operador `new` de JavaScript:

```
var gmap = new GMap2(document.getElementById("mapa"));
```

Al crear una nueva instancia de mapa, se especifica un nodo DOM en la página (normalmente un elemento `div`) como **contenedor** para el mapa. Los nodos HTML son secundarios del objeto `document` de JavaScript y obtenemos una referencia a este elemento mediante el método `document.getElementById()`.

El código mostrado anteriormente define una variable (denominada `gmap`) y asigna la variable a un nuevo objeto `GMap2`. La función `GMap2()` es un constructor que admite dos parámetros. El primero (obligatorio) hará referencia, tal y como hemos visto en el ejemplo anterior, al contenedor del mapa y será normalmente un elemento `div`. También se puede pasar un segundo parámetro opcional de tipo `GMap2Options`, que no es más que un array asociativo donde se determinan las características o propiedades que queremos que tenga el mapa. De esta forma se puede especificar, por ejemplo, el tamaño en píxeles con la propiedad `size` y el cursor que se debe mostrar mientras se está arrastrando el mapa con la propiedad `draggingCursor`.

Una vez creado el nuevo objeto del mapa, el siguiente paso es inicializarlo. Para ello se utiliza el método `setCenter()`, que requiere una coordenada `GLatLng` y un **nivel de acercamiento** o zoom. Es obligatorio enviar este método antes de llevar a cabo cualquier otra operación en el mapa (incluso la configuración de cualquier otro atributo en el propio mapa).

El objeto `GLatLng` proporciona la posibilidad de hacer referencia a ubicaciones del mapa dentro del API de Google Maps. Los objetos `GLatLng` se construyen pasando sus parámetros en el orden habitual en cartografía, es decir, primero la **longitud** y después la **latitud**.

El nivel de acercamiento define la resolución de la vista actual. En la vista normal de los mapas se pueden emplear los niveles de acercamiento entre el 0 (el nivel más bajo, que permite ver todo el mundo en un mapa) y el 19 (el más alto, que llega a mostrar edificios concretos). En este caso se obtendría una vista intermedia.

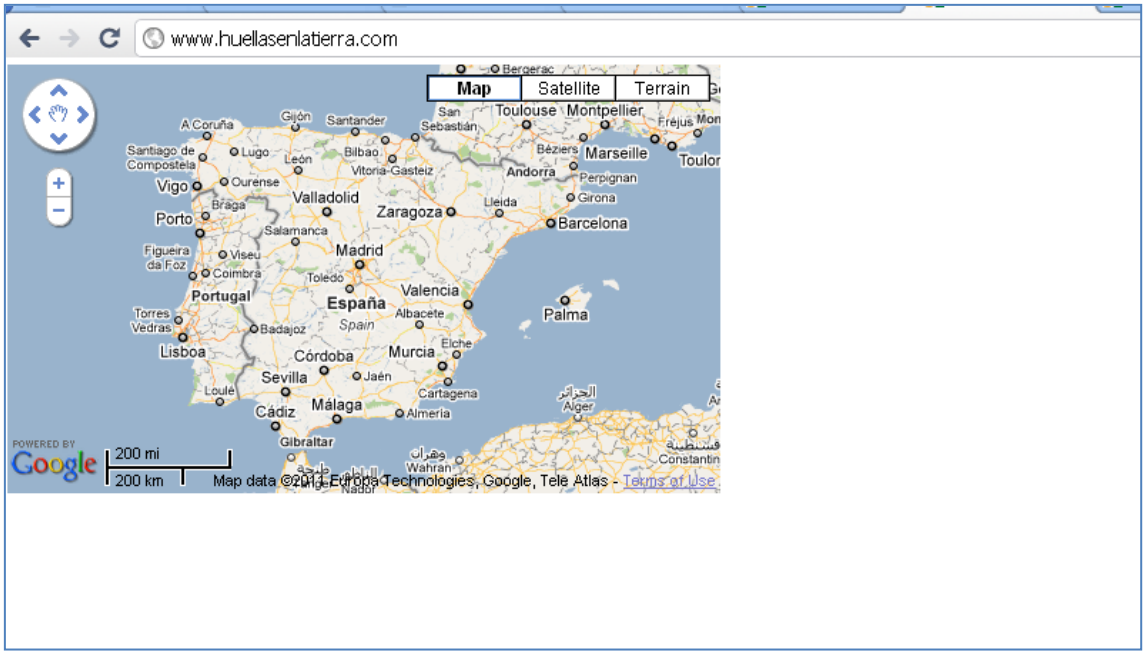
```
gmap.setCenter(new GLatLng(40.420088, -3.688810, 8);
```

En este ejemplo, centramos el mapa en la coordenada `(40.420088, -3.688810)`, correspondiente al centro de la ciudad de Madrid. Además, le aplicamos un nivel `8` de acercamiento.

Por último, y conociendo ya el código para colocar, crear e inicializar el mapa, lo único que nos quedaría por hacer con él sería cargarlo. Para ello empleamos el siguiente código HTML:

```
<body onload="inicializar()" onunload="GUnload()">
```

Mientras se procesa una página HTML, se externaliza el modelo de objetos de documentos (DOM) y las imágenes y secuencias de comandos externas se reciben e incorporan al objeto `document`. Para garantizar que nuestro mapa sólo se añada a la página cuando se cargue por completo, ejecutamos la función que hemos denominado `inicializar()`, que se encargará de crear, tal y como hemos explicado antes, el objeto `GMap2` cuando el elemento `<body>` de la página HTML ha recibido un evento `onload`. De este modo, evitamos un comportamiento impredecible y obtenemos más control acerca del modo y el momento en que se dibuja el mapa.



Google Maps en nuestra web inicial

El atributo `onload` es un ejemplo de **gestor de eventos**. El API de Google Maps también proporciona varios eventos que podemos detectar para determinar cambios de estado. Haremos referencia a ello más adelante. La función `GUnload()` es una utilidad diseñada para evitar **fugas de memoria**.

2. Controles

Los mapas contienen elementos de interfaz de usuario que permiten que el usuario interactúe con el mapa. Estos elementos se denominan **controles** y se pueden personalizar mediante subclases de la clase `GControl`. El API de Google Maps incorpora varios controles integrados que podemos emplear en el mapa. `GLargeMapControl` y `GSmallMapControl` son los controles de desplazamiento y acercamiento más grande y más pequeño respectivamente y se muestran en la esquina superior izquierda del mapa de forma predeterminada. Existen otros como `GScaleControl`, que indica la escala del mapa y el más utilizado, `GMapTypeControl`, que hace referencia a los botones que permiten al usuario alternar entre tipos de mapas. Hay cuatro tipos:

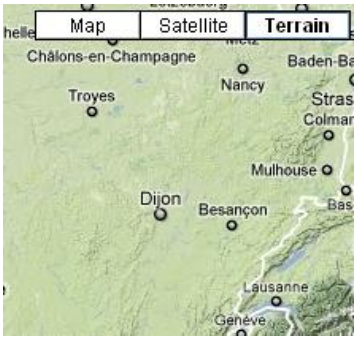
- `G_NORMAL_MAP` muestra los mosaicos predeterminados en 2D de Google Maps.
- `G_SATELLITE_MAP` muestra imágenes de satélite.
- `G_HYBRID_MAP` muestra una mezcla de mosaicos fotográficos y una capa de mosaicos para los elementos del mapa más destacados (carreteras, nombres de ciudades, etc).
- `G_PHYSICAL_MAP` muestra mosaicos de mapas físicos a partir de la información de relieve.

La forma más común para indicar el tipo de mapa es utilizar el método `setMapType()` pasándole como parámetro cualquiera de los tipos señalados antes.


GLargeMapControl


GSmallMapControl

GScaleControl



GMapTypeControl (G_PHYSICAL_MAP)

Para añadir controles al mapa se utiliza el método `addControl()`.

```
gmap.addControl (new GLargeMapControl());
```

Aún así, tenemos un método `getDefaultUI()` que devuelve los controles por defecto y que más tarde son añadidos al mapa con el método `setUI()`. Los controles por defecto son todos los comentados anteriormente y el tipo de mapa es `G_NORMAL_MAP` siempre que no le indiquemos lo contrario.

Las siguientes líneas de código muestran la inicialización del mapa `gmap` de nuestra aplicación:

```
var controlesPorDefecto = gmap.getDefaultUI();
gmap.setUI(controlesPorDefecto);
gmap.setCenter(new GLatLng(40.420088, -3.688810), 6);
gmap.setMapType(G_HYBRID_MAP);
```

En la última línea vemos que, al ser `G_NORMAL_MAP` el tipo de mapa por defecto, señalamos explícitamente que este sea `G_HYBRID_MAP`.

3. Eventos sobre el mapa

Los eventos del API de Google Maps se gestionan mediante funciones de utilidades en el espacio de nombres de `GEvent` para registrar los detectores de eventos. Cada objeto del API de Google Maps exporta una determinada cantidad de eventos con nombres. Por ejemplo, el objeto `GMap2` visto antes exporta los eventos `click`, `dblclick` y `move`, entre otros muchos. Cada evento se produce en un contexto determinado y puede pasar argumentos que lo identifiquen. El evento `mousemove`, por ejemplo, se activa cuando el usuario mueve el ratón sobre un objeto del mapa y transmite el objeto `GLatLng` de la ubicación geográfica en la que se encuentra el ratón.

Para registrar la notificación de estos eventos, se utiliza el método estático `GEvent.addListener()`, que toma un objeto, un evento que debe detectar y una función a la que se debe llamar cuando se produzca el evento especificado. Estas funciones reciben el nombre de **manejadores de eventos**. A continuación se muestra un ejemplo donde se crea un nuevo mapa, se centra en Madrid y se le asigna una función que muestra un mensaje de alerta cuando el usuario hace clic sobre él.

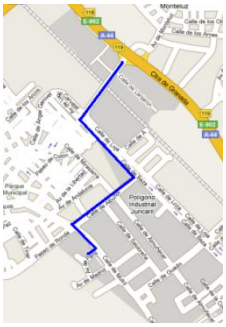
```
var gmap = new GMap2(document.getElementById("mapa"));
gmap.setCenter(new GLatLng(40.420088, -3.688810), 8);

GEvent.addListener(mapa, "click", function() {
      alert("Has hecho clic en el mapa");
});
```

4. Superposiciones

Las **superposiciones** son objetos del mapa vinculados a coordenadas, por lo que se mueven al arrastrar – siempre que el programador lo permita – o aplicar el acercamiento sobre el mapa. Las superposiciones representan objetos que se añaden al mapa para designar **puntos, líneas** o **zonas,** por lo que serán los elementos que darán la funcionalidad principal al mapa.

En nuestro caso, las huellas representan puntos. Así pues, este será el objeto principal de nuestro proyecto. Los puntos del mapa se representan mediante **marcadores** y suelen mostrar un icono personalizado. También se ha añadido la funcionalidad de mostrar al usuario determinadas fronteras, por lo que también se han utilizado zonas con una forma arbitraria representadas mediante **polígonos**. Existen otro tipo de superposiciones como las **polilíneas**, que representan un conjunto de puntos para formar una línea o un conjunto de líneas interconectadas en el mapa, aunque en nuestro proyecto no las hemos empleado explícitamente. Aún así, resulta evidente que los polígonos no son más que polilíneas cuyos puntos conforman un bucle cerrado que puede tomar cualquier forma.

Marcador


Polilínea

Polígono

Cada superposición implementa la interfaz `GOverlay` y se pueden añadir al mapa mediante el método `addOverlay` del mapa. Como es lógico, también se pueden eliminar con el método `removeOverlay`. El método `clearOverlays()`, por su parte, elimina todas las superposiciones del mapa y activa el evento `clearoverlays`.

4.1. Marcadores

Los **marcadores** son la base de nuestro proyecto ya que son los que identifican puntos en el mapa. De esta forma, cada huella representada en el mapa será un marcador y cada una de ellas tendrá sus propias características como puedan ser el icono, la coordenada y la información textual. Cada marcador es un objeto `GMarker` y su constructor utiliza `GLatLng` y un objeto `GMarkerOptions` como argumentos. El primero, como hemos visto antes, indica la coordenada en el mapa del marcador, es decir, dónde estará localizado. Esta es la característica principal del marcador y, por lo tanto, es un parámetro obligatorio. El segundo parámetro es opcional y determina, en forma de array, las otras características que puede tener el marcador como, por ejemplo, el icono (propiedad `icon` de tipo `GIcon`), el hecho de poder arrastrarlo o no con la propiedad booleana `draggable` o de poder hacer clic sobre él con la propiedad `clickable`, también booleana.

```
var icono = new GIcon();
icono.image = "iconos/1.png”;
var centroMadrid = new GLatLng(40.420088, -3.688810);
var huella = new GMarker(centroMadrid, {icon: icono; draggable: true});
```

En este ejemplo hemos visto como, en primer lugar, se crea un nuevo icono representado con la imagen `1.png` situada en el directorio `iconos`. También se crea una variable `GLatLng` llamada `centroMadrid` que determinará la ubicación del marcador `huella` tal y como vemos en el primer argumento del constructor `GMarker`. A dicho marcador también le asignamos el icono creado con anterioridad y la opción de poderlo arrastrar.

Existen muchas más características que pueden acompañar al marcador y, tanto las que hemos señalado como las demás, se pueden ir modificando en el código a medida que lo vayamos necesitando. Así, por ejemplo, en nuestro proyecto damos la posibilidad al usuario de fijar una huella cuando se haya guardado sobre ella toda la información y no desea cambiar su ubicación, de tal manera que deshabilitamos la opción de arrastre sobre el marcador y este queda inamovible. Esto lo hacemos con el método `disableDragging()` del marcador. Aún así, si el usuario desea modificar más adelante la posición de la huella o su información le damos la posibilidad de editarla y poder así cambiar su coordenada con el método `enableDragging()`.

Más métodos y propiedades de los marcadores:

El atributo `value` de un marcador, de tipo `String`, guarda la información relativa al mismo en forma de texto. Esta propiedad puede ser utilizada con un doble propósito si dicho `String` lo escribimos en formato HTML. Por un lado, tal y como hemos comentado, se encarga de almacenar cualquier tipo de dato que haga referencia al marcador y podremos acceder a cualquiera de ellos interpretando el código HTML mediante elementos DOM. A su vez, dicho atributo no sólo se dedicará a recopilar información del marcador sino que también nos ayudará, gracias a su formato HTLM, a diseñar la forma en la que se muestran los datos del mismo. Para enlazar este código HTML al marcador llamamos al método `bindInfoWindowHtml()` del marcador en cuestión, cuyo parámetro obligatorio será su propio atributo `value`. Aún así, como es lógico, podremos pasarle cualquier otro parámetro de tipo `String` y mantener su atributo `value` únicamente para recoger las características del marcador que estemos procesando.

Para mostrar toda esta información sería necesario crear un evento (por ejemplo, el evento *click* sobre un marcador) cuyo manejador llamase a la función `openInfoWindowHtml()` y a la que le pasaríamos como primer argumento el código HTML que deseamos mostrar. Para cerrar dicha ventana utilizaríamos la función `closeInfoWindow()`.

Ventana de información sobre un marcador

A continuación se muestra un extracto de código de nuestro proyecto correspondiente al proceso de creación de una nueva huella.

```
[…]
var _marcador = new GMarker(punto, {icon: icono });
_marcador.disableDragging();
_marcador.value = crearHtmlHuellaFija(idPOI,usuario,caracter,urlIcono,
                                      nombre,contenido, true);
_marcador.bindInfoWindowHtml(_marcador.value);

GEvent.addListener(_marcador, "click", function() {
      this.openInfoWindowHtml(this.value);
                  });

GEvent.addListener(_marcador, "dragstart", function() {
      this.closeInfoWindow();
                  });
```

En esta parte de código vemos cómo se crea un marcador dado un `punto` de tipo `GLatLng` creado con anterioridad y al que se le asigna un `icono` de tipo `GIcon` también conocido. Asimismo, deshabilitamos momentáneamente la opción de arrastre y le asignamos como `value` el `String` en formato HTML obtenido al ejecutar la función `crearHtmlHuellaFija()`, que recibe como parámetros variables obtenidas previamente y que, como se puede suponer, hacen referencia a datos informativos de la nueva huella. Una vez hecho esto, enlazamos con el marcador el código HTML obtenido antes a través de la función `bindInfoWindowHtml()`. Además, mostramos también dos eventos que asociamos al marcador. En este caso se generan el evento `click`, que, como vemos, permite abrir la ventana de información, y el evento `dragstart`, al que se le asocia un manejador que cierra dicha ventana cuando se inicia el proceso de arrastre sobre la huella.

Además de todos los métodos ya citados, existen otros muchos que pueden igualmente resultar de utilidad. Así, por ejemplo, es común utilizar el método `getLatLng()` que se encarga de devolver la coordenada del marcador, o `getIcon()` para obtener el icono definido en el constructor.

Eventos sobre un marcador:

Los marcadores en los que no puede hacerse clic o que no se pueden arrastrar son inertes, consumen menos recursos y no responden a ningún evento. De todas maneras, los marcadores están diseñados para ser **interactivos** y en nuestro proyecto, obviamente, no hemos desaprovechado esta particularidad. De forma predeterminada, todos los marcadores reciben eventos *click*, por ejemplo, y se suelen utilizar dentro de detectores de eventos para abrir ventanas de información.

En el punto anterior hemos explicado cómo utilizarlos y hemos visto que su uso no dista mucho del proceso que seguimos para asignar eventos al mapa. Lo único que habría que modificar es el objeto que se le pasa al método `GEvent.addListener()` que en este caso, como es lógico, sería el propio marcador en vez del mapa.

El administrador de marcadores:

En nuestro proyecto nos vemos obligados a utilizar un gran número de marcadores porque así lo exige la naturaleza y complexión de la página web. En este caso, representar cada huella o marcador como un objeto `GMarker` único puede reducir considerablemente la velocidad del procesamiento del mapa y provocar muchos defectos visuales, especialmente con determinados niveles de acercamiento. La utilidad del **administrador de marcadores** nos proporciona una solución para ambos problemas, permite una visualización eficaz de cientos de marcadores en el mismo mapa y ofrece la posibilidad de especificar los niveles de zoom en los que deseamos que se muestren los marcadores.

Un administrador de marcadores, por lo tanto, es un gran conjunto de ellos y en Javascript lo manejamos como un objeto `GMarkerManager`.

Podemos especificar varias opciones para ajustar de forma precisa el rendimiento del administrador de marcadores. Estas opciones se pasan mediante un objeto `GMarkerManagerOptions` que contiene, entre otros, el campo `maxZoom`, que especifica el nivel máximo de zoom supervisado.

```
var gmap = new GMap2(document.getElementById("mapa"));
var opciones = {maxZoom: 10};
var marcadores = new GMarkerManager(gmap,opciones);
```

Una vez creado un administrador, el siguiente paso es añadirle marcadores. `GMarkerManager` admite la adición de marcadores individuales con el método `addMarker()` o un conjunto con el método `addMarkers()`. Los marcadores únicos añadidos mediante `addMarker()` aparecerán de inmediato en el mapa siempre que cumplan los requisitos de la vista actual y el nivel de acercamiento especificado.

En cambio los marcadores que se hayan añadido mediante el método `addMarkers()` no se mostrarán en el mapa hasta que se llame de forma explícita al método `refresh()` de `GMarkerManager`, que añade al mapa todos los marcadores de la vista actual y la región de desplazamiento del borde.

El método `addMarkers()` cuenta con 3 argumentos:

- Un array de marcadores (de tipo `GMarker`).
- Zoom mínimo, de tal manera que los marcadores colocados en el mapa aparecerán en él si quedan dentro de la ventana gráfica del mapa y si el nivel de acercamiento es superior o igual al valor mínimo especificado.
- El tercer parámetro es opcional e indica el valor máximo de zoom, lo que significa que los marcadores se eliminarán automáticamente cuando el nivel de acercamiento en el mapa supere el valor especificado.

El método `addMarker()`, por su parte, tiene los mismos parámetros que `addMarkers()`, aunque el primero será un único marcador de tipo `GMarker`.

El la siguiente línea de código de nuestra aplicación se añade un array de marcadores denominado `PUBLICOS` al gestor de marcadores llamado `controladorMarcadoresPublicos`. Como el nivel de zoom mínimo especificado es igual a cero y el nivel máximo no se ha indicado, estos marcadores aparecerán siempre que se encuentren en el interior del gráfico del mapa.

```
controladorMarcadoresPublicos.addMarkers(PUBLICOS,0);
```

4.2. Polilíneas

Las **polilíneas** son muy útiles en Google Maps ya que pueden indicar al usuario, por ejemplo, el itinerario de ciertas carreteras o recorridos personalizados.

Una polilínea se dibuja en el mapa empleando funciones de dibujo vectorial del navegador si las hay o, si no, una superposición de imagen desde los servidores Google. Estos objetos se representan mediante la clase **`GPolyline`**, cuyo constructor tiene un parámetro obligatorio que es una matriz de vértices representados por su latitud y longitud, es decir, una matriz de elementos de tipo **`GLatLng`**. Hay otros argumentos opcionales como el color, que se indica en formato hexadecimal, la anchura de la línea en píxeles y el nivel de transparencia, que es un número entre 0 y 1.

Al igual que los objetos de tipo `GMarker` o `GMarkerManager`, los elementos `GPolyline` deben ser añadidos al mapa mediante el método `addOverlay()` de `GMap2` y

cuentan con una serie de funciones para facilitar su manejo. El método `deleteVertex()`, por ejemplo, elimina el vértice con el índice especificado en la polilínea y actualiza su forma debidamente. Otro método interesante es `getLength()`, que devuelve la longitud (en metros) de la polilínea a lo largo de la superficie de una Tierra esférica.

4.3. Polígonos

La utilidad de los **polígonos** se basa en la delimitación de zonas y, en un nivel más avanzado, en la representación de edificios y objetos 3D sobre el mapa. En nuestra aplicación se han usado para mostrar al usuario los puntos fronterizos de las Comunidades Autónomas de España y los de algunos países.

Los polígonos son objetos de la clase **`GPolygon`** y son muy similares a los de la clase **`GPolyline`** explicada anteriormente. La única diferencia radica en que se añaden las propiedades de relleno de la zona marcada, es decir, su opacidad y color.

Para añadir polígonos a nuestro proyecto no hemos utilizado esta clase explícitamente en el código Javascript, sino que hemos empleado un servicio que ofrece Google para que sus mapas puedan representar superposiciones simplemente leyendo un archivo KML, algo que pasaremos a mencionar en el siguiente punto.

5. Superposiciones KML y GeoRSS

El API de Google Maps admite los formatos de datos **KML** y **GeoRSS** para mostrar información geográfica.

Estos formatos de datos se añaden a un mapa con el objeto `GGeoXml`, cuyo constructor toma la dirección URL de un archivo XML de acceso público. Los marcadores de ubicación de `GGeoXml` se muestran como objetos `GMarker`, mientras que las polilíneas y los polígonos de `GGeoXml` se muestran como polilíneas y polígonos del API de Google Maps API.

Los objetos `GGeoXml` se añaden al mapa mediante el método `addOverlay()` ya citado y se pueden eliminar del mismo mediante `removeOverlay()`.

Hay que tener en cuenta que `GGeoXml` es un objeto modular dentro del API de Google Maps y no se carga completamente hasta que se utiliza por primera vez, por lo que sólo se debe llamar a su constructor cuando la página se ha cargado completamente. Normalmente esto se logra llamando al constructor `GGeoXml` en el manipulador `onload` de `<body>` que hemos visto con anterioridad.

En nuestro proyecto, para mostrar las fronteras de las Comunidades Autónomas españolas contamos con una variable `GGeoXml` que enlazamos en la función `onload` con un archivo *kml* localizado en nuestro servidor en el que se encuentran todos los puntos que marcan las fronteras y otra información relevante a la hora de mostrarlas en el mapa, como la opacidad y los colores.

```
var urlEspana = "http://www.huellasenlatierra.com/kml/provinciasEspana.kml";
var geoXmlEspana = new GGeoXml(urlEspana);
```

Una vez hecho esto, sólo nos queda añadir al mapa la superposición analizada en el archivo *kml*. Esto lo hacemos, lógicamente, en la función a la que se llama cuando el usuario desea ver las fronteras.

```
gmap.addOverlay(geoXmlEspana);
```

A continuación mostramos el archivo KML citado en el que, para simplificar su comprensión, sólo mostramos los puntos que identifican la frontera de la ciudad de Madrid. Como la cantidad de estos puntos es considerablemente alta, también hemos recortado muchos de ellos.

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://earth.google.com/kml/2.1">
<Document>
	<name>España</name>
	<Style id="fronteras">
		<LineStyle>
			<color>ff00ff00</color>
			<width>2</width>
		</LineStyle>
		<PolyStyle>
			<color>6a00ffff</color>
			<colorMode>random</colorMode>
			<fill>1</fill>
			<outline>1</outline>
		</PolyStyle>
	</Style>

	<Placemark>
		<styleUrl>#fronteras</styleUrl>
		<name>Madrid</name>
		<description>Frontera de Madrid</description>
		<Polygon>
			<outerBoundaryIs>
				<LinearRing>
					<coordinates>
-3.5398,41.164999,0	-3.536568,41.160364,0	-3.540575,41.150502,0
-3.533481,41.147129,0	-3.529258,41.140727,0	-3.511239,41.134205,0
-3.499638,41.123708,0	-3.4922,41.113496,0	-3.492441,41.107516,0
-3.489466,41.1047,0	.....	-3.5398,41.164999,0
					</coordinates>
				</LinearRing>
			</outerBoundaryIs>
		</Polygon>
	</Placemark>
```

```
</Document>
</kml>
```

En este extracto vemos cómo se especifica primeramente el estilo gráfico que veremos en el mapa, es decir, el color y ancho de las líneas que unen los puntos y el color, relleno y contorno que tendrán los polígonos. Los colores se representan en código hexadecimal, aunque cuentan con la peculiaridad de tener dos caracteres más al principio de la cadena para determinar la opacidad.

Para dibujar una superposición en el mapa, añadimos una etiqueta `<Placemark>` e indicamos que es un polígono con la etiqueta `<Polygon>`. Los puntos que forman la frontera los escribimos dentro de la etiqueta `<coordinates>`, tal y como vemos en el ejemplo, donde dichos puntos vienen representados por su latitud, longitud y altitud. Esta última medida, en nuestro caso, es siempre igual a cero ya que utilizamos sólo la señalización de las zonas en 2D. Observemos también que tanto el primer punto como el último son iguales. Esto debe ser así para acabar de cerrar el polígono. Si no fuera así, los valores referentes al color de relleno y a la opacidad serían inútiles.

Aplicación

1.A nivel usuario

En este primer apartado vamos a detallar la aplicación estudiándola desde el punto de vista de un usuario, empezando por el funcionamiento general de la aplicación y continuando por la interfaz que se encuentra el usuario al entrar en la web y el uso que puede dar a esta.

Interfaz de usuario

Tal como vemos en la imagen siguiente podemos dividir la web en tres zonas: una zona de botones en la parte superior, un mapa en la parte central y a la derecha contamos con un menú de acciones y una sección de registro de usuarios.

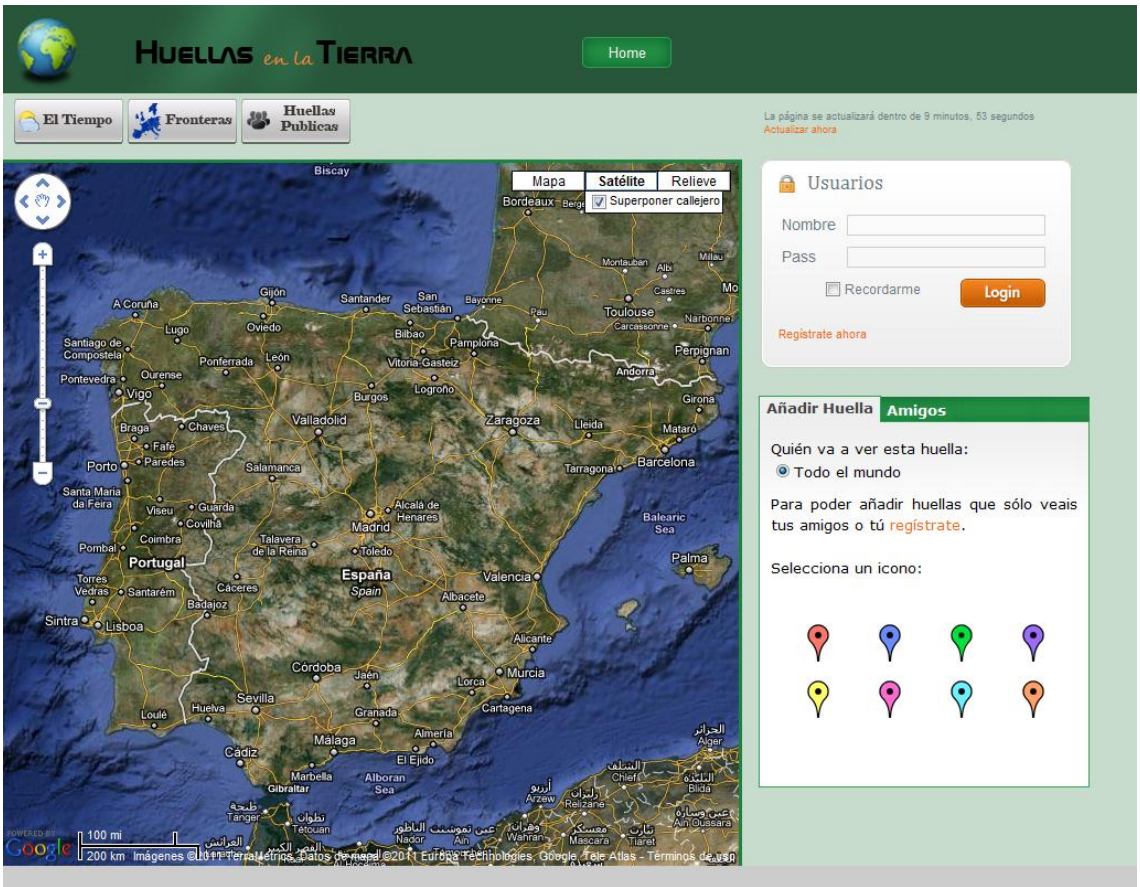


FOTO WEB (INVITADO)

La vista anterior nos muestra la web tal y como se la encontraría un usuario invitado, es decir, aquel que no se ha registrado e identificado como miembro de la web. Todo invitado tiene la oportunidad de probar ciertos contenidos de la

aplicación, pero sólo los usuarios registrados podrán hacer uso de todas sus funcionalidades. La siguiente imagen nos muestra la misma vista de antes para un usuario registrado.

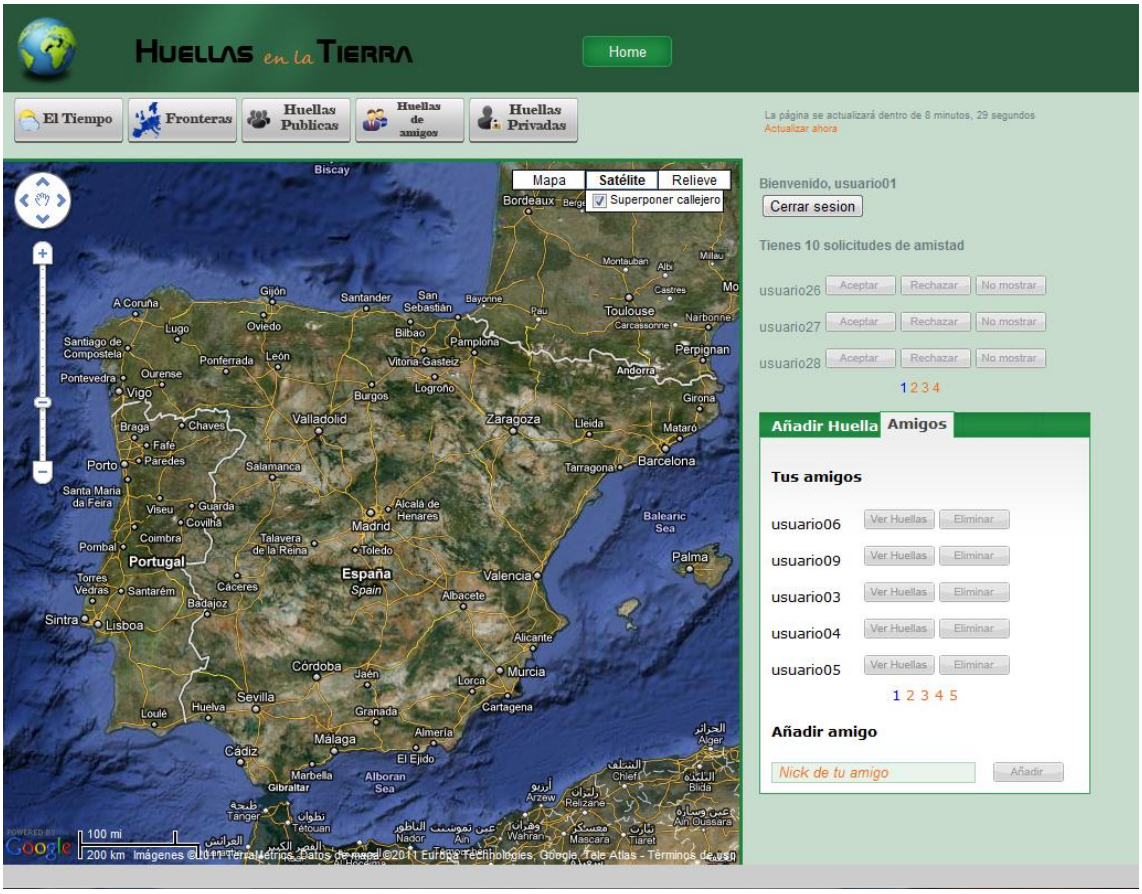


FOTO WEB (USUARIO REGISTRADO)

Botones de la zona superior

Los botones que encontramos en esta sección nos permiten controlar lo que se está viendo en ese momento sobre el mapa. Pudiendo activar o desactivar de manera independiente cada una de las opciones.

Como usuario invitado sólo veremos los tres primeros botones que nos permiten visualizar el tiempo, las fronteras y las huellas públicas. Sólo siendo un usuario registrado podremos visualizar las huellas de amigos y las privadas. El significado de cada botón se detalla más adelante en el funcionamiento general.

Mapa en la parte central

En un principio sólo se visualiza el mapa básico que obtenemos a través de la API de Google Maps, pero cuando interactuemos con las opciones que se van ofreciendo se irán añadiendo capas encima.

Desde un primer momento se ha tenido presente que el mapa es la parte más importante para el usuario, ya que la interacción de las huellas se hace sobre él y por tanto debe ser lo más grande que se pueda. Esto supuso un reto ya que en vez de usar un mapa de unas dimensiones estándar para todos los usuarios, se ha implementado de forma que las dimensiones del mapa se corrijan según la resolución de la pantalla de cada usuario, ajustándose así a todo el espacio que haya disponible.

Menú de acciones y registro de usuarios

El menú de acciones nos permite controlar la gestión de amigos, visualizar las huellas de estos, o añadir nuestras propias huellas. El registro de usuarios sirve para identificarse como usuario o registrar una cuenta nueva.

Funcionamiento general

El uso principal de la aplicación es que el usuario pueda guardar y compartir sus huellas con sus amigos. También se han añadido unas funcionalidades extras que permitirán poder ver las condiciones climatológicas en tiempo real con sólo examinar el mapa y otra que muestra los límites geográficos de cada zona.

Tipo de huellas

Existen tres tipos de huellas: públicas, para amigos y privadas.

Las huellas públicas son aquellas que están disponibles para todo el público, es decir que cualquier usuario puede ver. Al pinchar en el botón superior de “huellas públicas” veremos todas aquellas huellas que cualquier usuario haya creado de este tipo.

Las huellas de amigos únicamente las pueden ver los amigos que el usuario tenga agregados en su panel lateral y por supuesto el propio usuario.

Las huellas privadas son aquellas que sólo el usuario puede ver.

Gestión de huellas

Cuando un usuario crea una huella se convierte en el propietario de esta, de manera que sólo él pueda editarla o borrarla. En el caso de los usuarios invitados esto cambia, ya que no está identificado no puede ser el propietario de la huella, por tanto todos los usuarios pueden modificarla o borrarla.

Como a los invitados se les permite añadir huellas a modo de prueba esto puede generar muchas huellas inservibles, por eso al hacer el mantenimiento de la base de datos se pueden borrar las huellas de los invitados de forma programada: Cada 24 horas, cada semana o cada mes. Se haría dependiendo de la necesidad al ponerla en un entorno de uso real.

Crear huellas

Lo primero que tenemos que saber a la hora de crear una huella es quién vamos a querer que pueda verla. Para elegir el tipo de nuestra huella donde dice "Quién va a ver esta huella" podemos seleccionar si será pública, para amigos o privada.

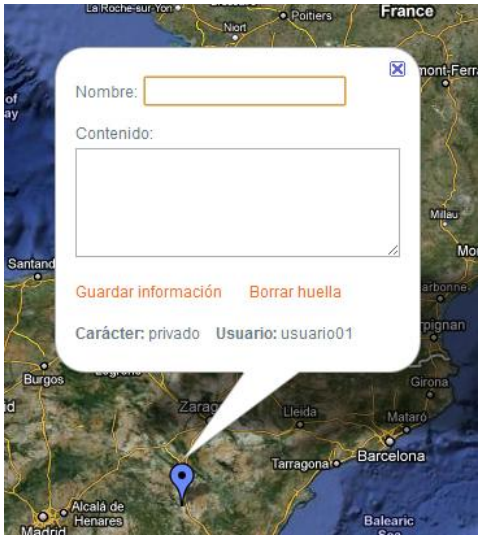
Una vez elegido el tipo es tan sencillo como elegir el icono entre uno de los que están disponibles en el menú de acciones, al pinchar sobre él podemos apreciar cómo se marca su fondo como seleccionado, y al pinchar en el mapa se añade una huella nueva. Podemos ver que la huella que hemos añadido está vacía, en el bocadillo de información que esta adjunto a esa huella hemos creado un formulario para poder introducir el nombre y el contenido que queramos, al darle

a guardar información la huella se fijará en el mapa de forma que ya no podemos moverla, y se guardarán los datos que tuviéramos metidos en los formularios.

Borrar huellas

Al pinchar sobre una huella, siempre se despliega un bocadillo que nos muestra el nombre y la información de dicha huella, y si somos los propietarios de ella nos aparecen dos enlaces debajo que nos permiten borrarla o editarla.

Editar huellas



Al pinchar en editar huella nos vuelve a aparecer un formulario como el inicial donde podemos introducir información. También nos permite moverla en el mapa para corregir su posición. Una vez hemos acabado se le da a guardar información para que se vuelva a fijar la huella. Al editarla no se puede cambiar la privacidad ni el propietario de la huella.

Gestión de amigos

En el menú de acciones de la derecha tenemos una pestaña “amigos”, en ella podemos encontrar todo lo que necesitamos para gestionar nuestros contactos.

Añadir amigos

Sólo tenemos que introducir el Nick del usuario que queremos que sea nuestro amigo, se le enviará una solicitud de amistad, cuando la acepte aparecerá automáticamente en nuestra lista de amigos. Si rechaza nuestra solicitud no nos aparecerá.

Ver Huellas

Al identificarnos como usuario registrado no sólo se cargan todas nuestras huellas sino también las de nuestros amigos, de esta forma sólo con pinchar en “ver huellas” al lado del Nick de nuestro amigo podremos ver sus huellas al instante. Cada amigo es independiente de los demás, por lo que podremos activarlos/desactivarlos de manera individual.

Eliminar amigos

Al lado del Nick de cada uno de nuestros amigos nos aparece un botón eliminar, que tiene una acción de confirmación asociada para evitar eliminar accidentalmente.

Solicitudes de amistad

Cuando otro usuario quiere ser nuestro amigo nos llega una solicitud de amistad, que aparece automáticamente encima del menú de amigos. Tenemos la opción de aceptarla, rechazarla y rechazar siempre.

Registro de usuarios

Este es el menú típico para la identificación de usuarios. Si el usuario no está registrado puede ir a otra pantalla que le permite crear una nueva cuenta, sólo tiene que pinchar en "registrarse".

2. A nivel programador

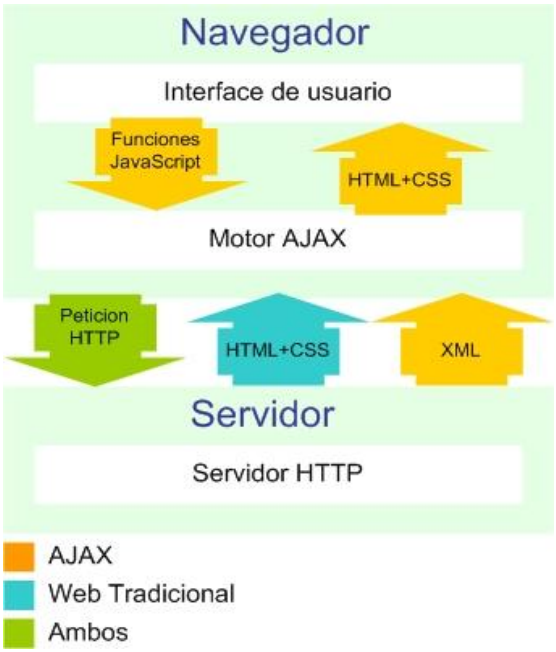
Antes de comenzar a detallar cómo se ha diseñado y especificado nuestra aplicación, haremos una breve mención a AJAX y su funcionamiento ya que esto nos permitirá comprender mejor los conceptos y las ideas que posteriormente desarrollaremos.

Breve introducción a AJAX:

La gran ventaja de **AJAX** es que es una tecnología asíncrona, lo que quiere decir que el usuario puede interactuar con la aplicación sin necesidad de refrescar la página. Todas las acciones solicitadas se cargan en segundo plano sin interferir con la visualización ni el comportamiento de la página. El lenguaje utilizado para efectuar las funciones de llamada AJAX es, normalmente, Javascript y el acceso a los datos se realiza mediante `XMLHttpRequest`, objeto disponible en todos los navegadores actuales.

El funcionamiento de AJAX, por lo tanto, se basa en el acceso "directo" del usuario a la aplicación que es enviada por el servidor en formato HTML, Javascript y CSS. El código Javascript de la aplicación solicita al servidor los datos que quiere mostrar y este ejecuta un código de lado del servidor y envía al navegador los datos en formato XML.

Cada vez que el usuario realiza una acción, la capa Javascript repite dicha acción de manera invisible al usuario y muestra los datos deseados.



Todas las aplicaciones realizadas con técnicas de AJAX deben instanciar en primer lugar el objeto `XMLHttpRequest` comentado antes, que es el objeto clave que permite realizar comunicaciones con el servidor en segundo plano, sin necesidad de recargar las páginas. La implementación de este objeto depende de cada navegador, por lo que es necesario discriminar de una manera simple en función del navegador en el que se está ejecutando el código.

```
if (window.XMLHttpRequest) {
    peticionAjax = new XMLHttpRequest();
}
else if (window.ActiveXObject) {
    peticionAjax = new ActiveXObject("Microsoft.XMLHTTP");
}
```

Los navegadores menos actuales implementan el objeto `XMLHttpRequest` como un objeto de tipo *ActiveX* y estos, por lo tanto, se irán por el segundo `if` del código mostrado. Los navegadores más recientes lo pueden obtener a través del objeto `window`.

Una vez obtenida la instancia del objeto `XMLHttpRequest`, el siguiente paso es preparar la función que se encarga de procesar la respuesta del servidor. La propiedad `onreadystatechange` de `XMLHttpRequest` permite indicar esta función directamente incluyendo su código mediante una función anónima o indicando una referencia a una función independiente.

Después de preparar la aplicación la respuesta del servidor, se realiza la petición HTTP al servidor mediante el método `open()`, en el que se incluye el tipo de petición (`GET` o `POST`), la URL solicitada (normalmente un archivo que se encarga de generar un documento XML) y un tercer parámetro para indicar si la petición es síncrona (`false`) o, como será habitual, asíncrona (`true`). Cuando se ha creado la petición HTTP, se envía al servidor mediante el método `send()`, en el que incluiremos los parámetros que determinan la información que se va a enviar al servidor junto con la petición HTTP.

```
peticionAjax.onreadystatechange = funcionRespuesta;
peticionAjax.open("GET", "archivo.php", true);
peticionAjax.send("nombre=Juan&soltero=si");
```

En este ejemplo, `archivo.php`, mediante código PHP, genera un archivo XML a partir de los parámetros `nombre` y `soltero` indicados en la función `send()` y que, como vemos, se separan mediante el carácter `&`.

La función de respuesta, en este caso `funcionRespuesta`, es la encargada de analizar el XML producido y terminar de procesar la acción deseada por el usuario. Esta función es ejecutada de forma automática una vez se ha recibido la respuesta del servidor, por lo que debe comprobar en primer lugar si se ha obtenido dicha respuesta mediante el valor de la propiedad `readyState`. Si se ha recibido alguna respuesta el siguiente paso consiste en comprobar que esta es válida y correcta. Para ello, se verifica que el código de estado HTTP devuelto (propiedad `status`) es igual a 200.

La respuesta del servidor sólo puede corresponder a alguno de los cinco estados definidos por las siguientes variables:

```
var READY_STATE_UNINITIALIZED = 0;
var READY_STATE_LOADING = 1;
var READY_STATE_LOADED = 2;
var READY_STATE_INTERACTIVE = 3;
var READY_STATE_COMPLETE = 4;
```

El contenido de la respuesta del servidor, por su parte, se puede recibir en forma de cadena de texto o en formato XML como en el ejemplo anterior. En el primer caso, la información se consigue mediante el método `responseText` y en el segundo la respuesta se obtiene mediante el método `responseXML` y el objeto devuelto se puede procesar como un objeto DOM.

A continuación mostramos un ejemplo sencillo en el que la función respuesta del servidor obtiene la información en formato XML producida por

`generaXML.php` y muestra al usuario las iniciales de un nombre completo mediante el método `alert` después de haberla analizado como un objeto DOM.

```
function accionUsuario() {
  if (window.XMLHttpRequest) {
    peticionAjax = new XMLHttpRequest();
  }
  else if (window.ActiveXObject) {
    peticionAjax = new ActiveXObject("Microsoft.XMLHTTP");
  }
  if (peticionAjax) {
    peticionAjax.onreadystatechange = funcionRespuesta;
    peticionAjax.open("POST", "generaXML.php", true);
    peticionAjax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    peticionAjax.send("nombre=Juan&apellido1=García&apellido2=Pérez");
  }
}

function funcionRespuesta(){
  if(peticionAjax.readyState == READY_STATE_COMPLETE) {
    if(peticionAjax.status == 200) {
      var documentoXML = peticionAjax.responseXML;
      var raiz = documentoXML.getElementsByTagName("respuesta")[0];
      var iniciales = raiz.getElementsByTagName("iniciales");
      alert(iniciales);
    }
  }
}
```

En este caso el documento XML producido tendría la siguiente forma:

```
<respuesta>
    <iniciales> … </iniciales>
    …
</respuesta>
```

La base de datos del proyecto:

A continuación explicaremos la estructura de la **base de datos MySQL** de nuestro proyecto, en la que se almacena toda la información relevante de los usuarios, los puntos de interés y los datos geográficos para la representación meteorológica. Para acceder a ella hemos utilizado la aplicación *phpMyAdmin* que ofrece el servidor y el programa *HeidiSQL*, que facilita bastante la visión de los datos.

Todas las tablas creadas utilizan la tecnología *InnoDB* ya que esta soporta transacciones y nuestro proyecto requiere continuamente de ellas. Además, *InnoDB* ofrece una fiabilidad y consistencia muy superior a *MyISAM*, la anterior tecnología de tablas existente.

Pasaremos ahora a detallar la estructura de cada una de las tablas que conforman nuestra base de datos. Antes de nada, señalaremos el significado de los iconos que pueden aparecer al lado de los campos:

PRIMARY KEY: Los campos que vengan señalados con este icono identifican la clave o claves principales de la tabla, de tal manera que dichos campos identifican de manera unívoca cada registro guardado.

UNIQUE: Indica que no pueden existir dos valores iguales.

FOREING KEY: Sirve para definir una clave foránea sobre un campo de la tabla, lo que quiere decir que sus valores hacen referencia a un campo de otra tabla. En nuestro caso, cuando se borra o actualiza un registro de la tabla cuyo campo es referenciado (tabla padre), también se realiza la misma acción en el registro coincidente de la tabla hija. Esto se consigue añadiendo las restricciones *ON DELETE CASCADE* y *ON UPDATE CASCADE*.

Comenzaremos explicando las tres tablas fundamentales de la aplicación:

1. USUARIOS:

#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default
1	**idUsuario**	INT	11	☐	☐	☐	AUTO_INCREMENT
2	nick	VARCHAR	20		☐		''
3	pass	VARCHAR	30		☐		''
4	email	VARCHAR	50		☐		''
5	codigoActivacion	VARCHAR	100		☐		No default
6	activo	TINYINT	4	☐	☐	☐	No default

El objetivo de esta tabla es almacenar todos los usuarios que se han registrado en el sistema. Para ello se guarda su *nick*, que será único, su contraseña (*pass*) y su correo electrónico (*email*), que también es único. La tabla asigna a cada persona registrada un número entero autoincremental (*idUsuario*) y será este el único valor que lo identifique en la base de datos. Además, tenemos fijado siempre un usuario genérico con *idUsuario* igual a 0 e "invitado" como valor del *nick*. Este usuario representa a la persona no registrada que usa nuestra aplicación. Cuando un usuario se registra, se crea una nueva fila en la tabla para él, pero no se activa en el sistema de inmediato (el campo *activo* vale 0) sino que se le designa un código (*codigoActivacion*) para confirmar el proceso mediante un link en el e-mail proporcionado (más adelante se especifica este procedimiento elaborado en el archivo *activacion.php*). Una vez confirmado, el usuario se activa (el campo *activo* pasa a valer 1) y el código de activación se cambia a 0 para anular el link anteriormente citado.

2. AMIGOS:

#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default
1	**idUsuario**	**INT**	**11**	☐	☐	☐	**0**
2	**idAmigo**	**INT**	**11**	☐	☐	☐	**0**
3	estado	TINYINT	4	☐	☐	☐	0
4	fechaSolicitud	DATE			☐		No default
5	fechaRespuesta	DATE			☑		NULL

En esta tabla existen dos filas (representadas por *idUsuario* e *idAmigo*) que hacen referencia al campo *idUsuario* de la tabla USUARIOS y que simbolizan la amistad (única) entre dos personas registradas. El campo *idUsuario* representa al usuario (usuario *A*) que quiere ser amigo del identificado en *idAmigo* (usuario *B*), por lo que es importante señalar que el orden de ambos campos es relevante. Por consiguiente, la amistad entre dos usuarios, en un principio, no tiene porqué ser recíproca.

Conocido, por tanto, el deseo de *A*, nos queda conocer el de *B*. Esto se indica en el campo *estado*, que puede tomar tres valores. Este campo vale 0 cuando se ha abierto la solicitud de amistad de *A* y se está a la espera de la respuesta de *B*. Si este acepta a *A* como amigo, *estado* pasa a valer 1. Si, por el contrario, la solicitud se rechaza, el registro de la tabla que identifica la amistad se borra, aunque puede darse el caso que *B* no quiera volver a ser "molestado" por *A* y en este lance la fila no desaparecería, sino que habría que tener en cuenta esta situación en la base de datos poniendo *estado* con valor 2, evitando automáticamente de esta forma la posible reincidencia de *A*. El campo *fechaSolicitud* indica el momento en el que se solicitó la amistad por parte de *A* y *fechaRespuesta* la fecha de resolución de *B* con respecto a su amistad con *A*. Con esto, si *B* acepta a *A*, *fechaRespuesta* indicaría el momento en el que comenzó su estado de amistad.

3. POI:

#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default
1	**idPOI**	**INT**	**11**	☐	☐	☐	**AUTO_INCREMENT**
2	posx	DOUBLE		☐	☐	☐	0
3	posy	DOUBLE		☐	☐	☐	0
4	idUsuario	INT	11	☐	☐	☐	0
5	nombre	VARCHAR	50		☑		NULL
6	contenido	VARCHAR	1250		☑		NULL
7	urlIcono	VARCHAR	100		☑		NULL
8	caracter	ENUM	'publico',...		☐		'publico'

Todos los puntos de interés creados por los usuarios se guardan en esta tabla, donde cada huella se identifica únicamente por un entero (*idPOI*). Los campos *posx* y *posy* indican su latitud y longitud respectivamente e *idUsuario* el

identificador del usuario que ha introducido la huella. Si este no está registrado, se le asigna el usuario genérico comentado antes (*idUsuario* igual a 0). Además, cada huella guarda una información textual representada por un *nombre* o un título y un *contenido* de 1250 caracteres como máximo. El campo *urlIcono* indica la localización del archivo imagen del icono en el árbol de directorios de nuestro servidor (ver siguiente punto). Por último, *carácter* determina si la huella es pública, protegida o privada.

Las tablas que describiremos a continuación se crearon con el objetivo de mostrar la información meteorológica y, por lo tanto, hacen referencia a la localización geográfica de todos los municipios españoles y, por ahora, de algunas de las localidades internacionales más conocidades e importantes. Todas ellas están relacionadas entre sí por un campo y se pueden representar de forma escalonada, ya que cada tabla identifica un nivel y es dependiente del superior (excepto el primero). El nivel más alto representa los continentes, el segundo los países, el tercero las provincias y el cuarto y último las localidades, que simbolizan el nivel geográfico más elemental en nuestra base de datos.

4. CONTINENTES:

#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default
1	**idContinente**	TINYINT	4	☐	☐	☐	AUTO_INCREMENT
2	nombre	VARCHAR	25		☐		No default

En esta tabla se almacenan los continentes existentes con su identificador único *idContinente* y su *nombre*.

5. PAÍSES:

#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default
1	**idPais**	SMALLINT	6	☐	☐	☐	AUTO_INCREMENT
2	nombre	VARCHAR	80		☑		NULL
3	idContinente	TINYINT	4	☐	☐	☐	No default

Aquí se recogen todos los países del mundo, en total, 212. La estructura es la misma, aunque se añade un campo más relacionado con la tabla anterior (*idContinente*) para indicar el continente al que pertenece.

6. PROVINCIAS:

#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default
1	**idProvincia**	**INT**	**11**	☐	☐	☐	**AUTO_INCREMENT**
2	nombre	VARCHAR	80		☐		No default
3	idPais	SMALLINT	6	☐	☐	☐	No default

La tabla de provincias sigue el mismo patrón que las anteriores. En la actualidad sólo recogemos las provincias españolas y las más relevantes a nivel mundial, aunque en un futuro se irán añadiendo las restantes.

7. COORDENADAS:

#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default
1	**idCoordenada**	**INT**	**11**	☐	☐	☐	**No default**
2	poblacion	VARCHAR	100		☑		NULL
3	latitud	FLOAT	10,6	☐	☐	☐	No default
4	longitud	FLOAT	10,6	☐	☐	☐	No default
5	altitud	FLOAT	10,6	☐	☐	☐	No default
6	idProvincia	INT	11	☐	☐	☐	No default
7	zoom	TINYINT	4	☐	☐	☐	0

Esta tabla representa el último nivel geográfico de nuestra base de datos y en ella se recogen, primeramente, los datos correspondientes a todas las localidades guardadas, es decir, su identificador (*idCoordenada*), nombre (*población*) y provincia a la que pertenece (*idProvincia*), que será un campo relacionado, obviamente, con la tabla de nivel superior. Además, al tratarse del último nivel, se almacena ya la ubicación geográfica de la localidad mediante su *latitud*, *longitud* y *altitud*, aunque esta última medida, de momento, no la hemos utilizado para ningún cometido. Sin embargo, gracias a la latitud y la longitud seremos capaces de mostrar en el mapa los iconos meteorológicos en la ubicación exacta de la población, además de conocer, a través del campo *idProvincia* y de las tablas anteriores, a qué provincia, país y continente pertenece sin repetir información, algo que podría facilitar la ampliación de funcionalidades en un futuro.

También contamos con un campo *zoom* para indicar cuál es el nivel máximo de acercamiento en el que queremos que se muestren los iconos de las localidades. De esta forma, evitamos que gran cantidad de iconos se amontonen en una zona reducida cuando el nivel de acercamiento del mapa es muy escaso. Por lo tanto, cuando esto ocurre, sólo hacemos que se muestren los iconos de las capitales de cada país (su valor de acercamiento máximo es alto). Por el contrario, si el nivel de acercamiento es mayor, la facilidad para diferenciar los iconos en cada punto es

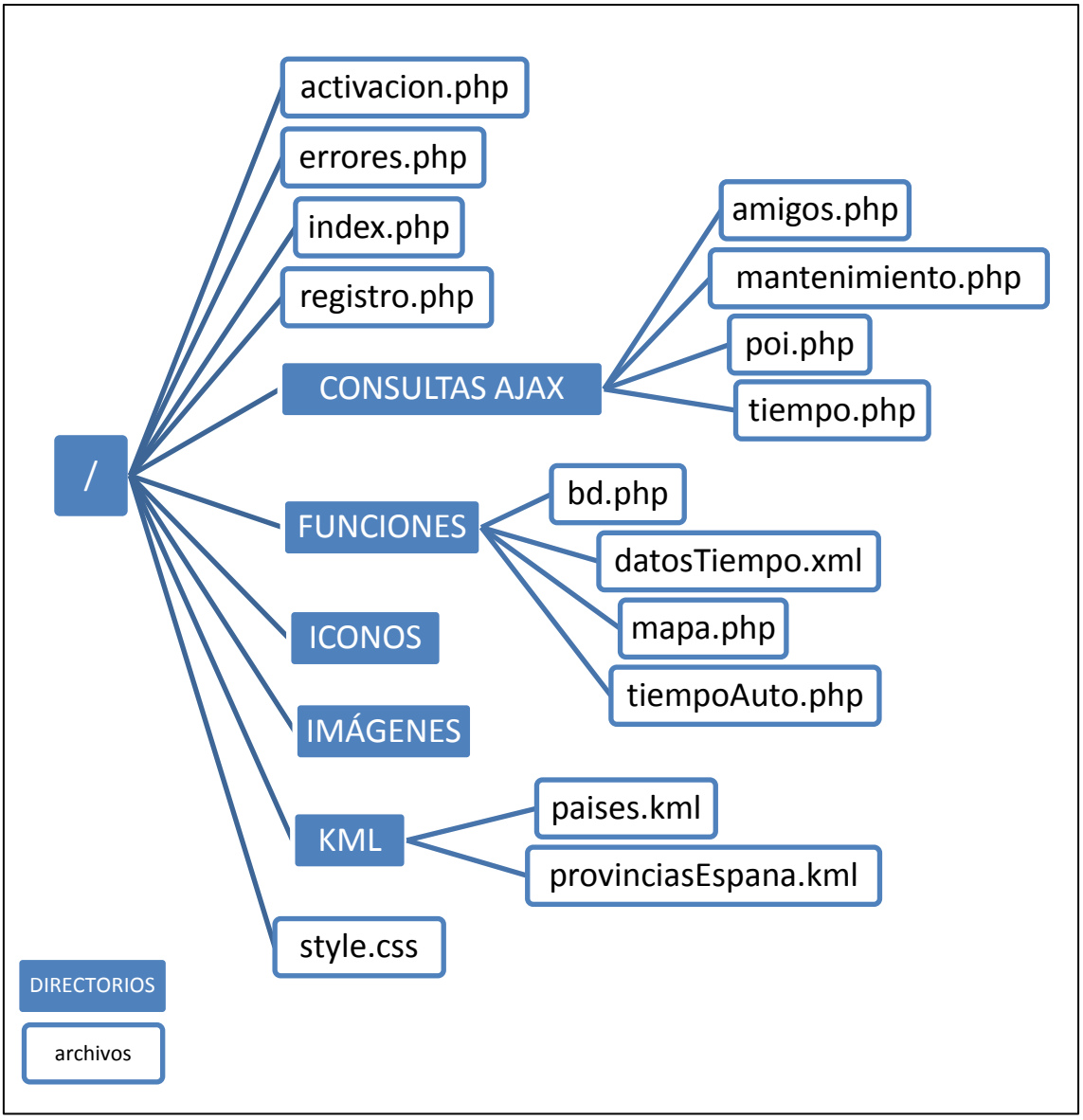
también mayor, por lo que se puede habilitar la aparición de los iconos de las localidades de menor interés. Por consiguiente, el nivel máximo de acercamiento que guardamos en la tabla para este tipo de poblaciones es menor.

Cabe destacar, por último, que, las localidades extranjeras actualmente almacenadas guardan el mismo nombre tanto en la tabla de provincias como en la de localidades (tabla COORDENADAS). Así, por ejemplo, la localidad de Berlín, perteneciente al continente Europa y al país de Alemania, pertenece también a la provincia de Berlín. Esto ocurre, generalmente, con todas las provincias como Madrid, Barcelona o Sevilla, aunque, en el caso de estas últimas, al pertenecer a España, también contienen otra gran cantidad de localidades. En el caso de Berlín - vista como provincia - sólo contiene, de momento, una localidad, que es la propia Berlín.

Como hemos comentado ya en alguna ocasión, esperamos que las funcionalidades de nuestra página web vayan extendiéndose poco a poco y que, por lo tanto, la base de datos de provincias y localidades vaya también ampliándose, de tal manera que lo que hemos comentado no sólo ocurra con las provincias y localidades españolas.

Estructura y funcionalidad de los archivos:

Una vez presentados anteriormente los conceptos básicos de AJAX y la estructura de nuestra base de datos, ya estamos en disposición de describir la **organización de archivos** que hacen posible el funcionamiento de la página web, para más tarde entrar a detallar cada uno de ellos en base a su estructura y funcionalidad.



Árbol de directorios de la aplicación

Tal y como vemos en el árbol de directorios, existen cuatro archivos *php* en la carpeta raíz, donde *index.php* es el archivo principal y el que lee el servidor al cargar la página. Los demás hacen referencia al registro de usuarios y al tratamiento de errores. El archivo *style.css* se ocupa de especificar el diseño de la web mediante clases e identificadores para las capas `div` del código HTML.

En el directorio referente a las consultas AJAX podemos encontrar cinco archivos. Estos serán los encargados de generar, a través de nuestra base de datos, los ficheros XML solicitados por las peticiones AJAX correspondientes y que más tarde las funciones de respuesta se ocuparán de procesar.

Por otro lado, el directorio de funciones cuenta con dos archivos fundamentales: *bd.php*, en el que encontraremos todas las funciones necesarias para la interactuación con la base de datos, y *mapa.php*, en el que se desarrollan todas las funcionalidades del usuario con el mapa y sus amistades. Además,

datosTiempo.xml es el fichero en el que se concentra toda la información meteorológica. El archivo *tiempoAuto.php* se encarga de mantener actualizado el fichero XML anterior, ya que es ejecutado automáticamente por el servidor cada hora (Cron Job).

En la carpeta de iconos se guardan las representaciones gráficas de los marcadores y la meteorología en formato *png*. Por otra parte, la carpeta de imágenes almacena todos los dibujos, gráficos y símbolos necesarios para un atractivo diseño de la página web.

Por último, en el directorio *KML* se reúnen los ficheros KML que Google se encargará de leer para mostrar las fronteras en el mapa y que, de momento, se mantienen inalterables.

A continuación, pasaremos a detallar el esqueleto y las funcionalidades de los archivos más relevantes.

activacion.php:

Este archivo contiene el código PHP necesario para terminar el proceso de registro de un usuario. Como veremos más adelante, el link enviado al correo electrónico del usuario después del proceso de registro (detallado en *registro.php*) tiene la siguiente forma:

http://www.huellasenlatierra.com/activacion.php?id=XXXXXX

Como vemos, cuando el usuario pulsa con el ratón sobre este enlace, se le dirige al archivo *activacion.php* que estamos estudiando. Podemos observar, a su vez, que en la URL se añade un parámetro *id*, cuyo valor es recogido mediante el método *GET*. Una vez hecho esto, se comprueba en la base de datos el usuario al que se le ha asignado dicho valor como código de activación para darle de alta en el sistema de manera definitiva.

El código fundamental de este archivo, obviando los detalles de diseño y otros aspectos necesarios de HTML, se muestra a continuación:

```
[…]
<?php
if (isset($_GET["id"])) {
        if (activarUsuario($_GET["id"])) {
                echo "Proceso de registro completado satisfactoriamente ";
                ?> <a href="index.php"> Volver </a> <?php
        }
}
?>
[…]
```

La función *activarUsuario* , que está contenida en el archivo principal de la base de datos (*funciones/bd.php*) toma el código de activación obtenido y, si se encuentra una fila en la tabla de usuarios con este código, le activa en el sistema modificando el campo *activo* correspondiente.

errores.php:

Generalmente, cuando se produce algún error en una página web, producidos sobre todo por accesos incorrectos a la base datos, se muestra un mensaje de error excesivamente brusco para el usuario. Este fichero se encarga simplemente de alterar estos avisos para que se presenten de una forma más amigable.

Para ello, establecemos una función de usuario `error_personalizado` que se ejecuta cuando se produce un error en tiempo de ejecución. Esta función acepta como parámetros obligatorios el tipo de error y el mensaje generado. En nuestro caso, añadimos otros dos parámetros opcionales para indicarle al usuario dónde se produjo exactamente. Para especificar qué función se ocupa de estos errores utilizamos `set_error_handler`:

```
<?php
function error_personalizado($tipo,$mensaje,$archivo,$linea) {
	echo "<p class='error'> Se ha producido un error en la línea $linea del
archivo $archivo. El error dice: $mensaje </p>";
}
set_error_handler("error_personalizado");
?>
```

La clase CSS **`error`** se define en el archivo *style.css*.

registro.php:

Este archivo es llamado por *index.php* cuando el usuario hace clic sobre el link de registro y tiene como elemento fundamental un formulario HTML donde el usuario introduce sus datos de registro. Aquí es imprescindible tanto el uso de expresiones regulares para comprobar la validez de los campos, como funciones de acceso a la base de datos para verificar si el nombre elegido o el propio e-mail del usuario ya existen en la tabla de usuarios.

Para el primer cometido hacemos uso de la función `ereg` de PHP, que recibe un parámetro en forma de expresión regular y comprueba si concuerda con la variable proporcionada como segundo parámetro. Así, por ejemplo, para comprobar la validez del e-mail hacemos lo siguiente:

```
if (empty($email)) {
	$errorEmail = "<p>¡Falta rellenar el email!</p>";
```

```
        $todoCorrecto = false;
}
elseif (!(ereg("^[^0-9][a-zA-Z0-9_]+([.][a-zA-Z0-9_]+)*[@][a-zA-Z0-9_]+([.][a-
zA-Z0-9_]+)*[.][a-zA-Z]{2,4}$", $email))) {
        $errorEmail = "<p>¡El formato del email no es correcto!</p>";
        $todoCorrecto = false;
}
```

En primer lugar, como vemos, se comprueba que el usuario no ha dejado en blanco el campo correspondiente al e-mail. A continuación, si dicho campo no es vacío, utilizamos la función **`ereg`** antes descrita para verificar que el formato de correo electrónico es el correcto.

Además, el anterior extracto de código nos permite añadir más conceptos fundamentales para este archivo de registro. En él observamos que para almacenar el e-mail hacemos uso de una variable PHP denominada **`$email`**. Esta variable la obtenemos mediante el método POST, ya que es este el método utilizado para recoger la información facilitada en el formulario y es el propio archivo `registro.php` el que se encarga de procesarla.

```
$email = htmlspecialchars($_POST["email"]);
```

Observemos que, en PHP, para recoger los datos mediante el método POST se utiliza el array asociativo `$_POST`. El contenido de este array es vacío la primera vez que se ejecuta la página ya que el usuario todavía no ha introducido ningún parámetro. Por eso, para discernir si se ha ejecutado o no esta página por primera vez, lo único que hacemos es comprobar si `$_POST` es vacío. Si no lo es significa que el usuario ya ha enviado sus datos a través del formulario y es necesario obtener dichos valores para comprobar su validez. La función `htmlspecialchars` de PHP convierte los caracteres especiales de HTML en texto plano. De esta forma evitamos que el usuario cause molestias estructurales en el diseño de la web.

Por lo tanto, una vez que el usuario ha pulsado en el botón *Enviar* del formulario, se vuelve a recargar la página, se obtienen los valores `$_POST` y si existe algún campo inválido, indicamos al usuario la causa del error aún con el formulario presente para que lo modifique. Además de un incorrecto formato, otro posible motivo de la invalidez de los datos es la existencia en la base de datos del *nick* o del e-mail escrito, por lo que el usuario tendría que añadir otro. Si, por el contrario, todos los datos proporcionados son válidos sólo nos queda enviar un mensaje al correo electrónico del usuario para que confirme su registro (mediante la función `mail()` de PHP). Para ello, tal y como se comentó anteriormente, empleamos un código de activación, cuyo valor lo obtenemos a partir de la función `uniqid()`. Esta función devuelve un identificador único con un prefijo basado en la hora actual en microsegundos y es el valor que asignamos al parámetro *id* de la URL enviada al usuario.

http://www.huellasenlatierra.com/activacion.php?id=XXXXXX

Una vez enviado el e-mail, el nuevo usuario se inserta en la base de datos con dicho código de activación, pero aún sin activar, y es el archivo *activacion.php*, como vimos antes, el que se encarga de finalizar el proceso de registro.

index.php:

Este es el archivo básico de diseño. En él se encuentran todos los elementos `DIV` que proporcionan una correcta distribución del espacio en la página principal. A todos estos elementos se les asigna un identificador para que su correspondiente parte gráfica o división pueda ser modificada dinámicamente mediante el modelo de objetos DOM en cualquier otro archivo.

Otra función básica de esta página es la del control y manejo de **sesiones** en PHP. Cuando un usuario se registra, nuestra aplicación inicia una sesión que permanece abierta hasta que él mismo decide terminarla (mediante el botón *Cerrar sesión*). De esta forma, si dicho usuario decide salir de la página web sin haberse desconectado previamente, cuando vuelve a entrar en ella más tarde no necesitaría registrarse de nuevo ya que su sesión continuaría abierta.

Aún así, el programador cuenta con diversas formas adicionales de destruir las sesiones sin que se haya producido el cierre de la misma por parte del usuario. Una de ellas es por la inactividad de este durante un cierto periodo de tiempo, aunque en nuestro caso, las sesiones finalizan cuando el usuario cierra el navegador. Por lo tanto, una sesión de usuario de nuestra aplicación permanece abierta siempre que el usuario no se desconecte mediante el botón *Cerrar sesión* ni cierre el navegador.

Las sesiones en PHP se inician mediante la función `session_start()`, que se encarga de crear un array asociativo `$_SESSION` en el que se irán almacenando todos los datos de sesión deseados. En el caso de nuestra aplicación, contamos siempre con la variable `$_SESSION["usuario"]` que guarda el *nick* del usuario registrado. Dicho array se destruye cuando la sesión finaliza.

Además, en este archivo contamos también con el manejo de cookies. En el cuadro correspondiente al *login* de usuarios damos la posibilidad de recordar al usuario su *nick* y contraseña una vez se haya *logueado* con éxito. Esto proporciona a toda persona registrada una forma ágil y eficaz de iniciar su sesión ya que su *nick* y contraseña aparecerán automáticamente en los campos correspondientes cuando inicie la aplicación sin necesidad de volverlos a introducir. Para ello, hacemos uso de la función `set_cookie()` para guardar una cookie en el navegador del usuario. En nuestro caso, utilizamos tres: el *nick* del usuario, su *contraseña*, y la propia opción de *Recordarme* (con valor *true* o *false*). La función mencionada,

además de la variable que queremos recordar, junto con su valor, utiliza un tercer parámetro que indica el momento (representado en segundos) en el que la cookie dejaría de existir. Como hemos comentado, las cookies se guardan únicamente en el navegador del usuario, por lo que si este tiene deshabilitada la opción de almacenar cookies, este proceso sería inútil. Por lo tanto, esta facilidad que proporcionamos al usuario depende únicamente de él.

En el caso de dejar en blanco la opción *Recordarme* nos veríamos obligados a borrar las tres cookies del usuario que hubieran podido ser creadas con anterioridad. Para ello, indicamos en el tercer parámetro antes aludido un instante de tiempo anterior al actual.

En el siguiente extracto de nuestro código mostramos la parte relativa a las sesiones y cookies de nuestra aplicación:

```
session_start();
[…]
if (isset($_POST["botonLogin"])) {
   $_SESSION["usuario"] = htmlspecialchars($_POST["username"]);
   if (isset($_POST["checkRecordarme"])) {
      setcookie("recordarme",true,time()+(60*60*24*7*365));
      setcookie("usuario",$_SESSION["usuario"],time()+(60*60*24*7*365));
      setcookie("pass",$_SESSION["pass"],time()+(60*60*24*7*365));
   }
   else {
      setcookie("recordarme",false,time()-1);
      setcookie("usuario",$_SESSION["usuario"],time()-1);
      setcookie("pass",$_SESSION["pass"],time()-1);
   }
}
```

Como podemos observar, una vez que el usuario ha pulsado el botón de *login* se guarda su *nick* en el array `$_SESSION` y si ha dejado marcada la opción *Recordarme* se crean sus cookies asociadas, que permanecerán en el navegador del usuario durante un año. En caso contrario, dichas cookies se eliminan. La función `time()` devuelve el momento actual en segundos e `isset()` comprueba si una determinada variable está o no definida.

FUNCIONES/bd.php:

Aquí se definen todas las funciones relacionas con nuestra base de datos, por lo que el código aquí reflejado es incluido en la mayoría de los demás archivos.

Para acceder a la base de datos hemos hecho uso de un método orientado a objetos simple y eficaz basado en la clase **MySQLi**. A continuación mostramos las funciones básicas de conexión y acceso.

```
function conectarBD() {
	$servidor = XXXXX;
	$bd = XXXXX;
	$usuario = XXXXX;
	$pass = XXXXX;
	$conexion = new mysqli($servidor,$usuario,$pass,$bd);
	if (mysqli_connect_errno()) {
		echo "No se ha podido establecer la conexión con la base de
datos: " . mysqli_connect_error();
		exit();
	}
	return $conexion;
}

function cerrarBD($conexion) {
	$conexion->close();
}

function ejecutarSQL($comando,$conexion) {
	$resultado = $conexion->query($comando);
	return $resultado;
}

function cerrarEjecucionSQL($resultado) {
	$resultado->close();
}

function dameFila($resultado) {
	$fila = $resultado->fetch_array();
	return $fila;
}
```

El resto de funciones incluidas en este archivo hacen referencia constantemente a las anteriormente citadas. Como muestra de ello, incluimos la función que se encarga de proporcionarnos el *nick* de un usuario dado su identificador único.

```
function dameNick($idUsuario) {
	$conexion = conectarBD();
	$resultado = ejecutarSQL("SELECT nick FROM usuarios WHERE
				idUsuario=$idUsuario",$conexion);
	$nick = "";
	if ($fila = dameFila($resultado)) {
		$nick = $fila["nick"];
	}
	cerrarEjecucionSQL($resultado);
	cerrarBD($conexion);
	return "$nick";
}
```

FUNCIONES/mapa.php:

Este es el archivo más amplio del proyecto puesto que aquí se incluyen todas las funciones de interactuación del usuario con la página web, es decir, el código Javascript que crea las distintas peticiones AJAX y sus correspondientes

funciones de respuesta. Hasta ahora contamos con un total de 15 peticiones relativas a los siguientes procesos:

1. Carga de las huellas públicas del usuario
2. Carga de las huellas protegidas del usuario
3. Carga de las huellas privadas del usuario
4. Carga de las huellas de los amigos (protegidas)
5. Inserción de una huella
6. Actualización de ubicación geográfica de una huella
7. Actualización del nombre o contenido de una huella
8. Eliminación de una huella
9. Carga de los amigos
10. Inserción de un amigo
11. Eliminación de un amigo
12. Aceptación de una solicitud de amistad
13. Rechazo de una solicitud de amistad
14. Rechazo constante de una solicitud de amistad
15. Carga del tiempo

Las peticiones de carga (1-4, 9 y 15) son creadas cuando la página se refresca, por lo que estas inicializaciones están incluidas en la función `onload`. El resto dependen de las acciones que el usuario realice en nuestra aplicación, de tal manera que si, por ejemplo, dicho usuario decide insertar un punto de interés, se creará la petición AJAX correspondiente (la número 5 de la lista) y se ejecutará el código insertado en su función de respuesta. Esta función, generalmente, trata de parsear el XML producido por el archivo referenciado en la función `open` de dicha llamada.

Para almacenar la información general (huellas públicas, meteorología, …) y la personal (amigos, huellas protegidas, …) hacemos uso de arrays.

En el caso de los puntos de interés contamos con tres vectores para cada tipo de huella, además de un cuarto para guardar las huellas protegidas de los amigos. Por cada uno de estos arrays, contamos además con el administrador de marcadores correspondiente donde se añadirán las huellas almacenadas en los vectores. De esta forma, tal y como explicamos anteriormente, para añadir todas las huellas públicas a su administrador de marcadores hacemos uso del método `addMarkers` en su función de respuesta (peticiones 1-4).

```
controladorMarcadoresPublicos.addMarkers(PUBLICOS,0);
```

Si el usuario hace clic sobre el botón de mostrar las huellas públicas, lo único que hacemos es llamar a la función `refresh()` de dicho administrador.

Las peticiones 5-8 se encargan de modificar los arrays de las huellas.

Para la gestión de amigos hacemos uso de un array de amigos y otro de solicitudes de amistad. Dichos vectores se rellenan en la función de respuesta de la petición número 9 y para alterar su estructura mediante inserciones o eliminaciones hacemos uso de las peticiones 10-14.

Para la meteorología (petición número 15) empleamos un vector bidimensional para distinguir el zoom del mapa en el que se muestran determinadas provincias. De esta forma, las provincias más relevantes se guardan en un array del vector principal y las menos relevantes en otro.

A continuación mostramos a modo de ejemplo el proceso para cargar los amigos del usuario:

```
function cargarAmigos() {
    peticionAjax9 = cargarAjax();
    if (peticionAjax9) {
       peticionAjax9.onreadystatechange = cargarAmigosRespuesta;
       peticionAjax9.open("POST", "consultasAjax/amigos.php", false);
       peticionAjax9.setRequestHeader("Content-Type", "application/x-www-
                                                        form-urlencoded");
       peticionAjax9.send("modificacion=no&usuario="+usuarioActual+"&amigo=noi
mporta&nocache=" + Math.random());
    }
}

function cargarAmigosRespuesta() {
       if(peticionAjaxCompleta(peticionAjax9)) {
              var documentoXML = peticionAjax9.responseXML;
              var raiz = documentoXML.getElementsByTagName("respuesta")[0];
              var cjtoAmigos = raiz.getElementsByTagName("amigo");
              for (var i=0; i<cjtoAmigos.length; i++) {
                     var nickAmigo =
cjtoAmigos[i].getElementsByTagName("nickAmigo")[0].firstChild.nodeValue;
                     var idAmigo =
cjtoAmigos[i].getElementsByTagName("idAmigo")[0].firstChild.nodeValue;
                     var estado =
cjtoAmigos[i].getElementsByTagName("estado")[0].firstChild.nodeValue;
                     if (estado == 1) AMIGOS.push(nickAmigo);
                     else if (estado == 0) SOLICITUDES.push(nickAmigo);
              }
              mostrarAmigos();
              mostrarSolicitudes();
       }
}
```

Los parámetros que se adjuntan en la función `send` son trasladados al archivo generador del XML, en este caso, *amigos.php* de la carpeta relativa a las

consultas AJAX. Vemos también como dicho XML es tratado más adelante en la función de respuesta `cargarAmigosRespuesta()` mediante elementos DOM.

Las funciones `mostrarAmigos()` y `mostrarSolicitudes()` se encargan de plasmar en la página los amigos del usuario y las solicitudes de amistad con las que cuenta respectivamente. Ambas cuentan con un sistema de paginación para dividir los amigos en el caso de existan muchos.

FUNCIONES/tiempoAuto.php:

Este es el archivo que se ejecuta cada hora en nuestro servidor a través de un Cron Job. En él se genera en formato XML toda la información relativa a la meteorología de cada provincia. Estos datos se obtienen a partir de dos fuentes. Para obtener las coordenadas de las provincias accedemos a nuestra base de datos y, a partir de su identificador, accedemos a su información del tiempo gracias a una llamada a la página web de *Meteored* mediante la siguiente url:

http://api.meteored.com/?localidad=XXX&affiliate_id=XXX

En esta url vemos que se usan dos parámetros. El primero corresponde al identificador numérico de la localidad o provincia. Esto nos ha obligado, lógicamente, a que los identificadores de nuestra base de datos coincidan con los mismos que almacena *Meteored* para cada una de las provincias. El segundo parámetro hace referencia a nuestro identificador como usuario registrado de *Meteored*.

La información recibida mediante la url mencionada es recibida por XML. En PHP la forma de acceder a ella es muy sencilla:

```
$file = simplexml_load_file
                ("http://api.meteored.com/?localidad=$id&affiliate_id=XXX")
```

A través de la variable `$file` seremos capaces de recibir todos los datos relativos a la provincia identificada con la variable `$id` obtenida previamente de nuestra base de datos.

El XML producido se almacena en el archivo *datosTiempo.xml* de esta misma carpeta y es el que lee nuestra página al cargarse (petición número 15 de la lista anterior).

```
    $file=fopen("datosTiempo.xml","w+");
    fwrite ($file,"$xml");
    fclose($file);
```

La variable `$xml` es el *string* que va guardando toda la información obtenida en formato XML. La manera de cerrar el código de este archivo es bien distinta a la que empleamos en los archivos de la carpeta de las consultas de AJAX que veremos a continuación. Mientras que en estos últimos terminamos con un simple `echo "$xml"` para que dicho *string* sea interpretado más tarde por las funciones de respuesta de AJAX, en éste generamos directamente el archivo XML puesto que, al ser llamado cada hora, debe existir continuamente en el servidor.

CONSULTAS AJAX/amigos.php:

En este archivo se genera el XML que más tarde es procesado por las funciones de repuesta relativas a la gestión de amigos (peticiones 9-14). Los parámetros que recibe, como hemos visto antes, se pasan en la función `send` del objeto referente a la petición AJAX mediante el método GET y, dependiendo de cuál sea dicha petición, se generará un XML u otro.

En todas ellas mandamos un parámetro *modificación* para indicar si se trata de un cambio de en la tabla de amigos (inserción de amistad o cambio de estado de una solicitud) o si, por el contrario, se refiere a una simple consulta.

En el primer caso (peticiones 10-14), el XML devuelto en la variable de tipo *string* `$xml` tiene el siguiente perfil:

```
<respuesta>
	<amigo>
		<existe> ... </existe>
		<movimientoAnterior> ... </movimientoAnterior>
		<esElMismo> ... </esElMismo>
		<pendiente> ... </pendiente>
		<rechazada> ... </rechazada>
		<idAmigo> ... </idAmigo>
		<nickAmigo> ... </nickAmigo>
	</amigo>
</respuesta>
```

En el caso de la inserción de una nueva solicitud de amistad, la etiqueta `existe`, por ejemplo, indica si el nick de la persona con la que el usuario quiere iniciar una nueva amistad existe o no en el sistema. Cada etiqueta tiene un significado diferente y sus valores harán que la página web se muestre o actúe de una forma o de otra.

Cuando simplemente queremos consultar los amigos de un usuario (petición número 9) el XML que se envía a su función de respuesta tiene la siguiente forma:

```
<respuesta>
	<amigo>
		<idAmigo> ... </idAmigo>
		<nickAmigo> ... </nickAmigo>
		<estado> ... </estado>
	</amigo>
	<amigo>
	...
	</amigo>
	...
</respuesta>
```

Como vemos, en este archivo se muestran todos los amigos del usuario. Cada uno de ellos contiene su identificador, su *nick* y el estado en el que se encuentra la amistad. Dependiendo de este último, el amigo se mostrará en la parte de solicitudes, en la de amigos o directamente en ninguna por haber sido rechazado siempre.

CONSULTAS AJAX/poi.php:

Este archivo genera toda la información relativa a los puntos de interés. El proceso que sigue no cambia con respecto al anterior. En este caso, el XML devuelto es de este estilo:

```
<respuesta>
	<poi>
		<idPOI> ... </idPOI>
		<posx> ... </posx>
		<posy> ... </posy>
		<usuario> ... </usuario>
		<nombre> ... </nombre>
		<contenido> ... </contenido>
		<urlIcono> ... </urlIcono>
	</poi>
	<poi>
		...
	</poi>
		...
</respuesta>
```

CONSULTAS AJAX/mantenimiento.php:

Aquí se añaden funciones de mantenimiento de la base de datos. Una funcionalidad que podemos encontrar en este archivo, por ejemplo, es la de la actualización de identificadores de las provincias para que coincidan con los de la página de *Meteored* en todo momento. Si se diera la circunstancia que dicha página cambiara algún identificador, nuestra aplicación se vería mermada a la hora de

mostrar el tiempo, ya que mostraría predicciones que no corresponderían con las localidades deseadas. El objetivo de esta función es que esto nunca suceda.

Este archivo no se ejecuta nunca salvo expreso deseo nuestro ya que suele contener funciones con un alto retardo de tiempo.

CONSULTAS AJAX/tiempo.php:

Este archivo no genera un XML, sino que devuelve uno ya generado previamente. Tal y como hemos visto antes, el archivo *funciones/tiempoAuto.php* es el encargado de construir cada hora el fichero XML con toda la información meteorológica. Dicho fichero, como también hemos señalado, es *funciones/datosTiempo.xml* y como todos los datos del tiempo se encuentran guardados ya en él, lo único que realiza este archivo es leer este XML ya generado y mostrar su contenido a la función de respuesta de la carga del tiempo (petición número 15).

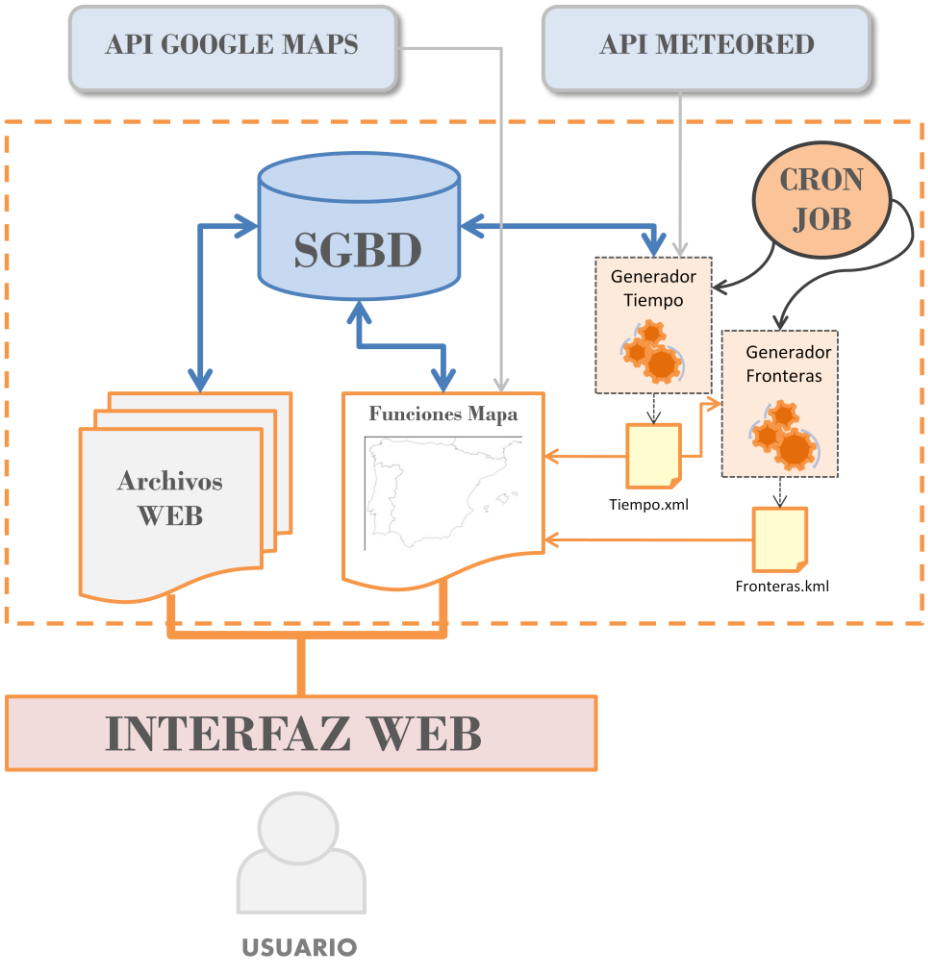
El código completo de este archivo, debido a su brevedad, lo mostramos a continuación:

```
<?php
	$ruta_fichero="../funciones/datosTiempo.xml";

	$contenido = "";
	if($da = fopen($ruta_fichero,"r"))  {
		while ($aux= fgets($da,1024)) {
			$contenido.=$aux;
		}
		fclose($da);
	}
	else {
		echo "Error: no se ha podido leer el archivo
<strong>$ruta_fichero</strong>";
	}

	header('Content-Type: text/xml');
	echo "$contenido";

?>
```

En este caso, como vemos, el *string* que devolvemos (`$contenido`) con la información en formato XML no se va generando poco a poco si no que se obtiene directamente a partir del fichero *funciones/datosTiempo.xml*.

Resumen

Para terminar de explicar el funcionamiento de nuestra aplicación y, a modo de resumen, añadimos un gráfico ilustrativo de la organización básica del proyecto:



Análisis de riesgos

1. Interacción asíncrona con el servidor

Una de las ideas de este proyecto era el aprender nuevas tecnologías sobre la marcha, pero la falta de conocimiento a veces lleva a nuevos problemas.

Desde un primer momento teníamos claro que queríamos que la interacción con la web fuese muy ágil y cómoda, lo que nos obliga a reducir el número de veces que refrescamos la página, ya que no estaría bien que cada vez que pinchamos en un elemento se recargue toda la web.

Tenemos que poder interaccionar con los elementos; aquí entra el lenguaje JavaScript. Ahora los elementos que usamos (huellas o amigos, por ejemplo) a su vez tienen que interaccionar con la base de datos. Esto lo pensábamos hacer combinando código PHP con el Javascript y aquí surge el problema: no se puede ejecutar código PHP una vez que la web esta cargada.

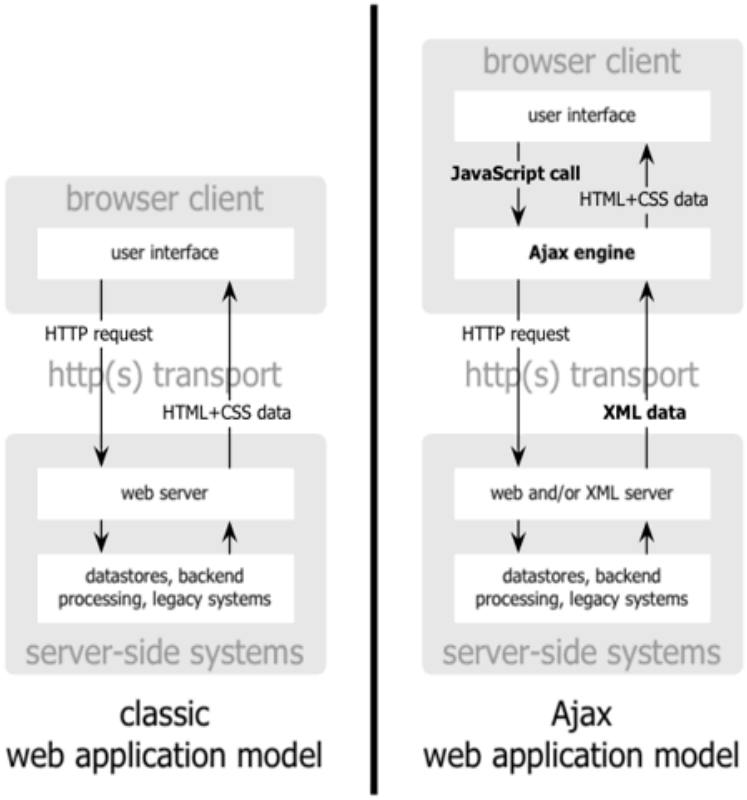
El código PHP se ejecuta del lado del servidor y Javascript en el lado del cliente, es decir, cuando el servidor da al cliente los datos, primero procesa el código PHP devolviendo directamente un HTML+CSS y luego es el cliente el que lo ejecuta en su ordenador, procesando en este momento el código Javascript, pero no pudiendo ejecutar código PHP para intercambiar información con la base de datos.

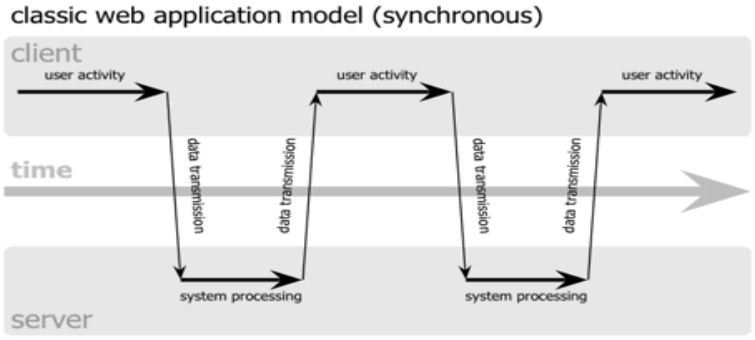
Con todo esto nos encontramos que no podíamos interaccionar de manera asíncrona (una vez que la web esta cargada) con el servidor.

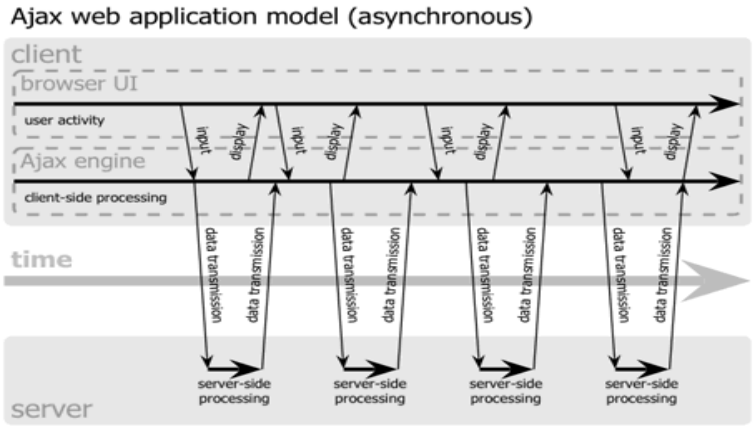
Solución:

Tras investigar mucho sobre el tema nos dimos cuenta que la solución pasaba por aprender AJAX. Ésta no es una tecnología en sí misma, en realidad es la unión de varias tecnologías de una forma nueva y sorprendente.

A continuación mostramos dos esquemas que nos permiten ver el funcionamiento de AJAX:


browser client
user interface
HTTP request
http(s) transport
HTML+CSS data
web server
datastores, backend processing, legacy systems
server-side systems
classic web application model
browser client
user interface
JavaScript call
HTML+CSS data
Ajax engine
HTTP request
http(s) transport
XML data
web and/or XML server
datastores, backend processing, legacy systems
server-side systems
Ajax web application model


classic web application model (synchronous)
client
user activity
user activity
user activity
data transmission
data transmission
data transmission
data transmission
time
system processing
system processing
server


Ajax web application model (asynchronous)
client
browser UI
user activity
input
display
input
display
input
display
input
display
Ajax engine
client-side processing
data transmission
data transmission
data transmission
data transmission
data transmission
data transmission
data transmission
data transmission
time
server-side processing
server-side processing
server-side processing
server-side processing
server

Ya habíamos comentado AJAX en el apartado de tecnologías usadas, así que sólo queda decir que es justo la solución que necesitábamos.

Este problema es el que ha supuesto un mayor retraso para el proyecto, ya que aprender AJAX con la ayuda de libros y tutoriales ha sido una parte complicada que ha llevado un tiempo que no estaba planeado.

2. Tiempo de carga

Para la implementación del proyecto se ha usado un servidor local. Esto resulta muy cómodo ya que es más rápido trabajar con él desde nuestro propio ordenador, sin necesidad de usar un cliente ftp cada vez que queramos subir o bajar archivos.

El problema es que los tiempos de carga de la web en el servidor real son mucho mayores que los del servidor local que usamos para el desarrollo de la web. Con esto nos encontramos que nuestra versión de la web que apenas tardaba en cargar tiene un tiempo de carga de unos 30-40 segundos, algo que era completamente inaceptable.

Solución:

Al entrar en la web se cargan una gran cantidad de contenidos, tanto nuestras huellas, como una lista de nuestros amigos (de cada uno de los cuales también hay que cargar las huellas), las solicitudes de amistad, el tiempo, etc… Es normal que tarde un poco en cargarse, pero tras investigar paso por paso nos dimos cuenta que es la carga del tiempo la que dispara los segundos de espera.

En este caso el tiempo tardaba tanto porque necesitaba pedir a la página de *Meteored* (*http://api.meteored.com*) la predicción del tiempo para cada una de las provincias, y gestionar la información resultante de todas ellas para mostrarla en el mapa.

Una vez localizado el problema hay que encontrar la solución. Lo que hemos hecho es crear un archivo XML que hace de intermediario en el proceso. Cada cierto tiempo se ejecuta automáticamente un archivo PHP, mediante un Cron Job, que pide los datos a *Meteored* y genera un archivo XML con toda la información del tiempo adjunta a cada punto del mapa.

De esta forma, en vez de acceder a *Meteored* cada vez que un usuario carga la web, se accede al archivo de información directamente. Con esto reducimos drásticamente el tiempo de carga volviendo a cifras de espera aceptables.

Diario de progresos

Este es un resumen del progreso del proyecto desde que no era más que una idea hasta su entrega y presentación.

Octubre

Recolección de ideas e investigación en general.

Noviembre

1) Introducirnos en todo el tema referente a dominios, hosting, etc. para saber cómo crear nuestra web y donde alojarla.

2) Creamos dominio y servidor.

3) Decidimos las tecnologías a utilizar, buscando información sobre cada una de ellas, con cursos, tutoriales, etc: Javascript, PHP5, CSS, MySQL, además de HTML.

4) Comenzamos a mirar las funciones de Google.

5) Planificamos el proyecto conforme al tiempo disponible: (puntos 1, 2, 3 y 4 hasta Navidad-enero 2011 aproximadamente, a partir de ahí empezar con los puntos 5,6 y 7)

Planificación proyecto:

1. Diseño e implementación de la página web (maquetado e interfaz principal)
2. Mostrar mapas con puntos de interés (POIs) (sólo POIs públicos sin registro de usuario)
3. Mostrar iconos del tiempo (sin especificar el área que abarca)
4. Mostrar mapas con registros de usuario y POIs con permisos.
5. Ampliaciones: Áreas de provincias (combinado con meteorología)
6. Mejora de interfaz gráfica de la web y optimización general.
7. Memoria del proyecto.

6) Decidimos la estructura de la base de datos, es decir, las tablas que utilizaremos, los campos y sus características, relaciones, etc, además de la estructura interna de los archivos y las carpetas (clases, funciones, imágenes, iconos, ...)

7) Elegimos el diseño de la web, cómo se mostrará la información dependiendo del usuario, registro del mismo, ...

8) Empezamos a estudiar las tecnologías elegidas. Las primeras que vamos a estudiar son HTML y PHP.

Diciembre

1) Continuamos con la fase de estudio, terminamos con el HTML, seguimos con el PHP y empezamos a su vez con Javascript.

2) Crear la base de datos y todas las tablas que tenemos pensado utilizar. Hemos creado instancias de prueba que más tarde eliminaremos.

3) Intercalamos el tiempo de estudio con la implementación de la web.

4) Encontramos el problema del sincronismo con la base de datos, investigamos y encontramos la solución en AJAX.

5) Tutoriales de AJAX.

6) Aprendemos las nociones básicas de cómo usar la API de Google Maps.

7) Empezamos a mostrar las huellas en el mapa.

Enero

1) Avanzamos en el estudio de la API de Google Maps.

2) Aumentamos la funcionalidad de las huellas que se muestran (añadir, editar, borrar, guardar,…)

3) Introducimos los privilegios de propietario para cada huella.

4) Terminamos con el registro de usuarios para poder identificarse.

5) Control de cookies y sesiones para los usuarios.

Febrero

1) Implementamos la gestión de amigos (añadir/borrar amigos, visualizar una lista de nuestros amigos, ver sus huellas, etc)

2) Estudiamos y aprendemos a usar la API de *Meteored*.

3) Comenzamos con la implementación del tiempo (conexión con *Meteored*, leer datos del tiempo)

4) Creamos diarios de progresos que usaremos a la hora de desarrollar la memoria.

Marzo

1) Mejora de la interfaz (Mapa extensible, página de registro de usuarios, botones interactivos,…)

2) Continuamos con la implementación del tiempo (introducir datos en la base de datos)

Abril

1) Añadimos la funcionalidad de las solicitudes de amistad, para obtener confirmación del amigo al que añades.

2) Terminamos con la implementación del tiempo (leer información de la base de datos, procesarla y mostrarla en el mapa)

3) Empezamos a investigar cómo conseguir y dibujar los límites de las provincias en el mapa. (Investigamos los KML)

4) Empezamos a escribir la memoria.

Mayo

1) Nos encontramos con el problema del tiempo de carga de la web (Buscamos la raíz del problema, investigamos posibles soluciones y encontramos una: los “Cron Jobs”)

2) Investigamos sobre los Cron Jobs, aprendemos a usarlos y los introducimos en nuestro proyecto para mejorar el tiempo de carga de la web (reimplementando parte de la gestión del tiempo)

3) Comenzamos la implementación para mostrar los límites provinciales.

4) Avanzamos en el desarrollo de la memoria.

5) Testeo para encontrar y solucionar errores.

Junio

1) Terminamos y entregamos la memoria.

2) Terminamos con la implantación de los límites provinciales.

3) Terminamos y entregamos el proyecto.

4) Preparamos la presentación final del proyecto.

Bibliografía

- Lee Babin. *Beginning AJAX with PHP: From Novice to Professional,* APress, 2007

- Andre Lewis, Michael Purvis, Jeffrey Sambells, Cameron Turner . *Beginning Google Maps Applications with Rails and Ajax: From Novice to Professional*, APress, 2007

- Javier Eguíluz Pérez. *Introducción a AJAX*, 2008.

Páginas web consultadas:

- Google Maps API Family: *http://code.google.com/intl/es-ES/apis/maps/index.html*

- Programación web: *http://www.programacionweb.net/cursos/curso.php?num=2*

- Wikipedia: *http://es.wikipedia.org/wiki*